

DRA

RESEARCH-STUDY OF A SELF-ORGANIZING COMPUTER

by
Mario R. Schaffner

(NASA-CR-140599) RESEARCH-STUDY OF A
SELF-ORGANIZING COMPUTER Final Report
(Massachusetts Inst. of Tech.) 268 p HC
\$8.50

N75-10715

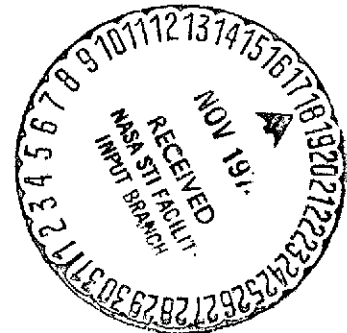
CSSL 09B

Unclas
63/60 17145

for the
National Astronautic and Space Administration
Contract NASW - 2276
Final Report

Massachusetts Institute of Technology
Cambridge, Mass. 02139 USA

July 1974



P R E F A C E

The title of the report deserves some explanation. The sponsoring Agency, the Guidance, Control and Information System Division of NASA, Washington, D.C., was interested in knowing about a hardware system programmed in the form of finite-state machines that was developed at the Smithsonian Astrophysical Observatory, and gave this contract with this title. The opportunity has been taken to study further and document the approach used. The result of the work can be properly labeled as an "organizable" computer. The capability of this computer to be organized is such that undoubtedly it would significantly facilitate the establishment of "self-organizing" systems, as soon as proper programming systems are added to it. However, no work could be made, in the limits of this contract, on self-organizing programs, although determined programs are amply documented.

"Self Organization" is not a clearcut notion. All complex systems have some degree of self-organization. However, the interpretation to be assumed here is that taken in the context of artificial intelligence: the development of means for performing given tasks, in relation to an environment. Essential to this approach is the establishment of criteria for evaluating performance. In this sense, self-organization has attributes of thinking. Turin (1950) in his "Can a machine think?" conjectured that a possible computer (a projection of the Manchester computer of that time), furnished with enough memory and programs, could produce results undistinguishable from those of human thinking. This shows that a self-organizing system has two main components: an organizable physical part, and a programming part. Different emphasis can be given to either of the two parts. This report presents the organizable part in the form of a programmable hardware and its programming language. The program part for the self-organization can be considered as a continuation of the present work.

PRECEDING PAGE BLANK NOT FILMED
TABLE OF CONTENTS

PREFACE	page iii
Chapter 1. MOTIVATION FOR A NEW APPROACH	1
1.1 Unsatisfactory Aspects in the Use of Computers	1
1.2 The Compiler Approach	5
1.3 The Approach of the Language Directed Computer	8
1.4 A Global Approach	12
Chapter 2. WHY ABSTRACT MACHINES?	15
2.1 Inferences from Psychology	15
2.1.1 Introduction	15
2.1.2 Developmental structures	16
2.1.3 The spatiotemporal frame	18
2.1.4 Words and imagery	20
2.1.5 Concluding remarks	26
2.2 Modeling and Representation	27
2.3 Automata Theory	33
2.3.1 Introduction	33
2.3.2 Functions and structures	34
2.3.3 Formalizations	35
2.3.4 Automata and languages	36
2.3.5 Connections with psychology	36
2.3.6 Interchangeability between program and machine	38
2.4 Cellular Spaces and Computing Structures	41
2.5 Inferences from Neurology	46
2.5.1 Introduction	46
2.5.2 A look at the central nervous system	48
2.5.3 Inferences from the central neural system	55
2.5.4 The McCulloch-Pitts correspondence	59

2.6	Outline of the Approach Taken	63
2.6.1	Synopsis, part 1	63
2.6.2	Function and structure	67
2.6.3	Synopsis, part 2	70
Chapter 3.	THE FORMULATION OF A SYMBOLIC SUBSTRATUM	73
3.1	Preliminaries	73
3.2	Elementary Machines (FSM Automaton)	78
3.2.1	Symbolic formulation	78
3.2.2	Structural formulation	79
3.3	Compounded Machines (CPL Automaton)	82
3.3.1	Symbolic formulation	82
3.3.2	Structural formulation	82
3.3.3	Discussion	86
3.4	Comparison with Other Formal Systems	88
3.4.1	Comparison with commonly formulated finite-state machines	88
3.4.2	Comments in respect to Turing machines	90
3.4.3	Considerations in regard to cellular spaces	92
3.4.4	Considerations in regard to formal languages	94
Chapter 4.	THE ISOMORPHIC IMPLEMENTABLE SUBSTRATUM	99
4.1	General Structure	99
4.1.1	Interface with the environment	99
4.1.2	Program allocation	101
4.1.3	The automatic flow of data	102
4.2	Implementation	104
4.2.1	The programable network	105
4.2.2	The programable memory	109
4.3	Discussion	110
4.3.1	Pipelining	110
4.3.2	Addressing	111

4.3.3	Parallelism	113
4.3.4	Computer architecture	114
4.4	The CPL 1 Processor	117
4.4.1	Factual information	117
4.4.2	Description	118
Chapter 5.	THE PROGRAMING LANGUAGE	121
5.1	Introduction	121
5.1.1	The role of the programing language	121
5.1.2	Preliminaries on the user language	125
5.1.3	An experiment in applying modes of thinking to computer features	127
5.2	The User Language	130
5.2.1	Structure of a program and its representation	130
5.2.2	Input prescription I	134
5.2.3	Data transformation F	135
5.2.4	Transition function T	143
5.2.5	Routing R	149
5.3	An Implentation of Machine Language	153
5.3.1	Outline	153
5.3.2	The format of the card	154
5.3.3	The morphology of the words	156
5.4	Discussion	158
5.4.1	Functional and technical characteristics	158
5.4.2	Similarities with other programing languages	168
Chapter 6.	COMPARISON OF PROGRAMS	173
6.1	Introduction	173
6.1.1	Available works on program comparison	173
6.1.2	Criteria of comparison	174
6.1.3	Description of the data used in the tables	177

6.2	Real-Time Processing of Weather Radar Signals	178
6.2.1	Structural characterisation of radar-echo patterns	182
6.2.2	Distribution of precipitation intensity	189
6.2.3	A compounded program	193
6.2.4	Recording of Weather echoes	196
6.2.5	Measurement of statistical characteristics of weather echoes	198
6.2.6	Measurement of the dispersion of short means	202
6.2.7	Real-time data handling	205
6.2.8	Measurement of the antenna pattern with solar noise	208
6.2.9	R.A.D.A.R.	211
6.2.10	Real-time numerical models	214
6.2.11	Computation of the Fast Fourier transform	218
6.3	Real-Time Processing of Meteor Radar Signals	223
6.3.1	Recognition and recording of faint meteors	223
6.3.2	Experiments of strategies	229
6.3.3	Noise measurements	231
6.4	Exploratory Programs	232
6.4.1	Numerical model of the dynamic of a fluid	232
6.4.2	Analysis of echo-pattern turbulence and movement	236
Chapter 7.	CONCLUDING REMARKS	243
7.1	The Essence of the Approach	244
7.2	Results Obtained and Extrapolations	246
7.3	Topics for Further Study	249
REFERENCES		253

Chapter 1

Motivation for a New Approach

1.1 UNSATISFACTORY ASPECTS IN THE USE OF COMPUTERS

Every year we become accustomed to seeing computers increase their speed, their capacity, and their endowment of automatic procedures. Reliability is not of concern any longer; on the contrary, we have come to expect infallibility. The cost-performance ratio decreases continuously. In sum, computers do not present technological problems. However, there are some unsatisfactory aspects in their use, as outlined in the following.

In modern life, computers are becoming almost as numerous as automobiles, and the feeling develops that computers are going to be an indispensable companion for many everyday activities. However, we observe that while the majority of automobile users drive their automobiles themselves, a significantly smaller portion of computer users do their own programming directly; as a matter of fact, a new profession has developed for the operation of these machines. Note, moreover, that the data we have to process, and the dynamic systems we have to consider in driving an automobile are much more complex than the very elementary mechanization of a currently typical computer program. In the course of our analysis we will find an interpretation for the difference in interaction that occurs with automobiles and computers (section 2.1.4, page 24).

Others (see for instance Sammet, 1969) use a different analogy. It has been said that if the telephone companies had not gone to dial telephoning, then every woman between the ages of 20 and 50 would have been forced to become a telephone operator in order to keep up with demands. Similarly, at the rate computers and their applications are developing, it may be necessary for vast numbers of people to become programmers. In fact, programmers

may well have to outnumber significantly the number of people who have problems to solve, inasmuch as the problems themselves become more and more complex.

Clearly we need to deal with computers directly, just as everybody today dials directly his telephone connections, and almost everybody drives his own car rather than obtaining the service of a professional driver. Naturally each user would like to do this with maximum application to his problem and with minimum attention to the annoying intricacies of the computer itself.

In order to obtain a general view of this situation, independent of particular applications and computers, we shall try to identify the essential, common protagonists involved in the use of a computer. If a computer is used, obviously it is for having executed some process conceived by a user; therefore, there will always be, as a starting point, a process in some user's mind. We do not elaborate on this for the moment, but indicate it as point A in Fig. 1.

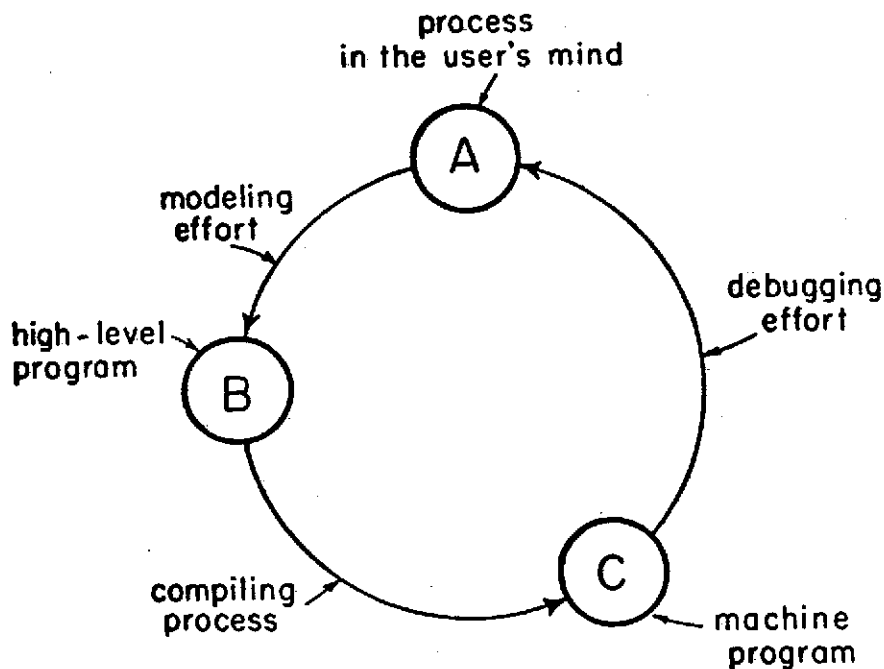


Fig. 1. Process transformations in the use of a computer.

With the same generality there will be also a package of information that actually controls the computer hardware for it to produce the desired process. All general-purpose computers manufactured today are digital machines that respond to organized sequences of instructions written in a given language and in an established format. Thus, we can identify the above package of information with the so-called object or machine program. We indicate this as point C in Fig. 1.

The machine languages of all commercially available computers are conceptually very simple but quite inappropriate for human users to express their problems. This makes impractical the production of machine programs directly by the user. Customarily an intermediate language is used, the so-called high-level programming language, such that the user can express in it a specific problem with reasonable ease, and the obtained description can be automatically transformed by the computer itself (in a preliminary run, under the control of the so-called compiler program) into a machine program for that specific problem. Thus there is a third protagonist: the program in high-level language, or source program; we indicate this as point B in Fig. 1.

Point B represents a concrete description (as a rule in the form of formal sentences) of a process. Essential characteristics of it are: (1) to be complete, in the sense that no further work is necessary on the part of the user in regard to the process per se (obviously further work might be expected for interfacing the process into a larger system); (2) to be translatable into point C automatically without human intervention. At this point it is irrelevant if in this transformation additional intermediate languages are used.

The activities involved in the use of a computer can be visualized as the interconnection of these three points. At the start, the process in the user's mind has to be modeled in terms of the elements of the programming language. Depending on the type of process and on the language used, this modeling may appear straightforward or as mental acrobatics. It is an effort that takes sometimes several hours and sometimes several months to perform. We indicate this activity with an arrow labeled "modeling effort" in Fig. 1. Then the source program (point B) is automatically transformed into the object program (point C). We use the term transformation rather than translation.

tion because typically the structure of the object program has little resemblance to the structure of the source program. If a good compiler is available, the user is completely relieved from the difficulties of this transformation. However, the development of the compiler constituted an effort at some previous time; it is well known that compilers are the hardest part in computer manufacturing, and that they make the system management complex. In Fig. 1, this transformation is symbolized by an arrow labeled "compiling effort". Finally, in the debugging phase, the actual actions made by the computers have to be related to the original image of the process that the user had in mind. It is well known that for nontrivial processes, debugging is the longest part in the programming activity. We symbolize this part in Fig. 1 with an arrow connecting C to A labeled "debugging effort".

We can now relate the inconveniences mentioned at the beginning of this section with the activities symbolized by the arrows in Fig. 1, and figuratively relate their length with the amount of effort involved. In the present approach to computers, A is not analyzed, C is maintained essentially at the level it happened to be in for the first computers, and most of the effort is devoted to finding the best location for point B. From this arises the proliferation of programming languages.

The conjecture is here made that the best overall solution cannot be obtained by maintaining this approach. Point B cannot be very close to point A; characteristics (1) and (2) mentioned for the source language constitute limiting constraints. For instance, we cannot use spoken languages because of the difficulties of automatic interpretation and translation. We cannot use a set of mathematical expressions because they do not form a complete, autonomous description of a process, in the sense that the connections between them and the interpretation of the symbols used need to be supplied. On the other hand, B cannot be too close to C, for instance, by using macro-assembler languages, because the modeling would be too cumbersome. One may think that for each specific application an optimum point B can be determined, but a large multiplicity of languages is undesirable and expensive.

In regard to debugging, the situation is no simpler. We leave aside here the theoretical question of proving the correctness of a program. For every complex new program, it is a fact of reality that some unexpected problem will occur. If the problem is unforeseen, the actual behavior at point

C must be directly analyzed against the requirements at point A, regardless of the debugging facilities available. One may think to minimize the occurrence of these events by providing all sorts of automatic aids; but in this case (given the distance between C and A), the aid system may be more complex than the computer itself; moreover, it is not obvious that the effort of interpreting them in the unforeseen cases will not nullify their benefit.

In the visualization of Fig. 1, one can think that all problems would be eased if point A and C were closer. All three arrows (symbols of the efforts) would be shortened. The ideal would be to shrink Fig. 1 into a single point. Such an approach requires a reconsideration with a fresh view of points A and C. Before going further, we will analyze in some more details the present approaches in the next two sections; the discussion then will be resumed in section 1.4.

1.2 THE COMPILER APPROACH

The automatic transformation of the source program into the object program by means of a compiler program is the basic approach of today's computers, and has made possible the present explosion of computer applications. Its rationale is to give users a high-level language appropriate for their problems, to optimize computer hardware in terms of the available technology, and to provide suitable software systems for connecting that hardware with that language. Given the different characteristics and requirements of a human being and a machine, the approach to treat them separately seems the most logical one. The unlimited power of language for expressing human thought is well known; thus this choice of form of expression is indeed universal. Studies in natural languages develop generative and recognizing procedures for increasingly larger subsets of them; studies in artificial languages develop increasingly richer systems; automata theory gives increasingly new insight into the structure of languages. As a consequence, the expectation develops that a formal language of sufficient power to be appropriate for general use, and automatically translatable, will in the future completely solve the problems of this approach.

To gain a better insight into this approach, let us expand Fig. 1 with more pertinent details. The user expresses the activities he has in mind in the form of stereotyped sentences; on the one hand this requires a remodeling and often a deformation of what he has in mind, and on the other

hand it permits him to prescribe with minimal means the very complex (invisible) activities that the computer has actually to do. This is the great advantage of present programming languages in the context of present computers. The compiler, in turn, has to interpret these minimal expressions and then reconstruct all the corresponding complex computer activities. This work can be indicated diagrammatically as in Fig. 2. The source program, constituted by strings of symbols with a certain flexibility as appropriate for the user, receives a lexical pass to be transformed into rigid strings of other symbols more appropriate for the machine manipulation. These strings are then subjected to a syntactic analysis for recognizing their grammatical structures and reducing them into explicit, complete, local forms. Next, a first machine-code generation is produced. Because of the unsatisfactory

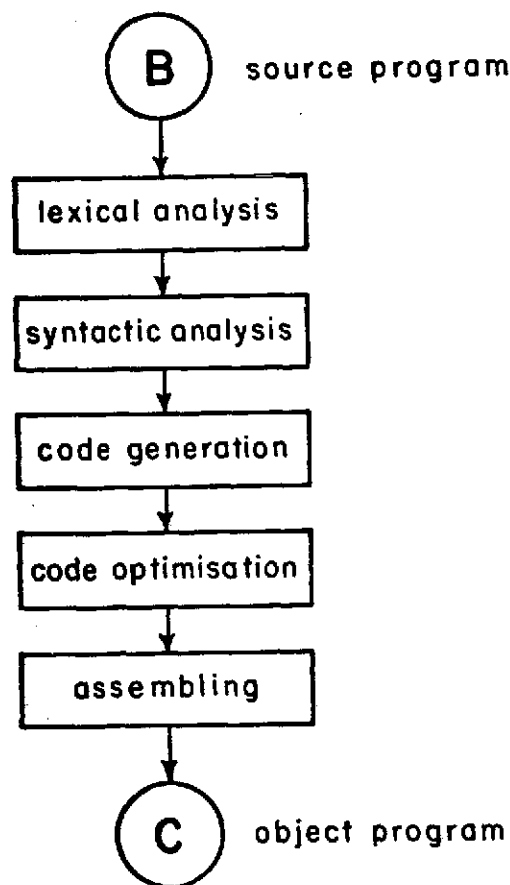


Fig. 2. Schematic sequence of operations in a program transformation from the user to the machine language.

object program that would result at this point, an optimizing phase follows which eliminates redundancies, changes the position of instructions in order to reduce their occurrences, chooses more efficient processes, and attempts to apply the several programming tricks. Finally, the actual object program is assembled, with assignment of explicit addresses, choice of registers, and ordering of the controls.

The power and potential of the approach of automatic language translation is well known and documented. The interest here is in comparing alternative approaches to it. We can observe that naturalness and flexibility in the programming languages clash with the complexity of the compiler. The more sophisticated the compiler is, the more mysterious the actual computer behavior appears to the user, with consequent dependency of the man on the machine, rather than vice versa. So far, no one programming language has proved sufficiently efficient for all types of problems to discourage the use of the other languages. On the contrary, the feeling develops that a single language would freeze the knowledge and techniques of a particular moment, blocking the benefits of successive developments. In fact, there has been a continuous development of languages each more appropriate in a broad range of problems and made more effective by software packages for each class of applications. The proliferation of programming languages that has occurred can be seen in Sammet (1969).

One of the problems of the proliferation of languages is the number of compilers that become necessary for the automatic translation of all users' programs for all types of computers. Suppose that there are 33 languages in use and 33 types of computers; for making each computer capable of work with each language, the computer community should possess $33 \times 33 = 1089$ compilers. For each new language and for each new computer, 33 new compilers have to be developed. This fact was recognized very early, and the possible advantage of a common intermediate language was pointed out (Strong et al. 1958). If all the above 33 programming languages can be translated to a hypothetical intermediate language (UNCOL), and programs in this language can be translated into machine programs of the 33 computers, the community would need only $33 + 33$ half compilers (half because the intermediate language would be halfway between a user's and a machine language). Each new language and each new computer would require the development of only

one-half new compiler. In spite of this appealing situation, such an intermediate language has not been developed (Steel, 1961a,b). Dennis (1971) faces this problem with a new insight, in terms of a common base for programming languages, simultaneously, and in accordance with general systematic guidelines for computer architectures.

Another problem of the compiler approach is the difficulty of obtaining efficient object programs, in comparison, for instance, with carefully hand-coded programs. Another inconvenience is that for each program modification an entire new compilation should be made. Direct interaction with the computer, in the presence of compilers, requires complex software systems; a similar situation occurs in regard to debugging. For many unusual problems, and typically when real-time processing is involved, compilers are not helpful, and the user has to proceed with the cumbersome preparation of the programs in machine language.

From the viewpoint of the study described in this report, we wonder about the amount of program manipulation (often exceeding the actual problem processing), and the complexity of the software systems required in this approach. The compiler approach tacitly presupposes that phrase-structure languages are the most appropriate means for communicating with a computer; the validity of this assumption will be questioned in chapter 2. Finally, we note that the roles and the difficulties of compilers may change significantly if the computer hardware takes a different approach.

1.3 THE APPROACH OF THE LANGUAGE DIRECTED COMPUTER

As a consequence of the problems encountered in the approach described in the previous section, the question rises recurrently whether computers constructed to execute directly programs written in the user programming language can lead to a more efficient overall system, especially in view of the technological development that are continuously occurring.

At the same time, every systematic approach that is attempted for constructing computers illustrates the wide range of designs that are possible and the different levels of abstraction that can be assumed as visible architecture. Examples of these studies can be found in Iliffe (1968), Kilburn et al. (1968), and Bijoner (1971). In this situation, one

thinks spontaneously going from the bottom up in Fig. 2, in the sense of implementing directly more and more of the program transformations, up to the direct hardware implementation of the user language.

These questions will appear again and again until a satisfactory proof is given that one or the other approach is preferable. On this subject the imaginary situation described by McKeeman (1967) is relevant:

It is an accident that digital computers are organized like desk calculators - with somewhat worse luck we might have taken the Turing machine as our model.

Now, if one of the users of an automatic digital Turing machine were to suggest revolutionary additions such as a large random access memory and a special command to add the contents of two memory cells, we could expect to find him attacked on various counts. A design engineer would complain that the additions were ad hoc and destroyed the essential simplicity of the Turing machine. Besides they would make the machine ten times more expensive to manufacture. Another user would wistfully agree that the additions were clever and nice but he couldn't afford the expense of reprogramming his entire library of Turing machine programs: a working group would publish a list of accepted standards that were in jeopardy. And finally it would be pointed out that next year's Turing machine would be twice as fast.

The disgruntled user would, of course, take refuge in higher languages. He would use a compiler that would, when it saw the symbol '+', generate the necessary 175 Turing machine instructions required to add two adjacent bit patterns on his tape.

As time went on, the user might console himself with progress: the discovery of a new 167 instruction add routine; hyper-Turing tapes where the bits are recorded in frames 9 bits wide; the appearance of a pipeline micro-parallel Turing machine which, under special circumstances, could execute 27 simultaneous Turing machine operations; and finally, the Ultimate - two time-shared Turing machines working on the same tape.

The solution of some problems would still be beyond reach and the federal government would allocate funds for a really parallel effort - 100 Turing machines arranged in a 10-by-10 array sharing 20 tapes on a grid, 10 across the rows and 10 down the columns. Since the machine could be shown to be potentially 100 times as fast for some problems, the best programmers in the world would stand in line to use it, thereby insuring its success by contemporary standards.

The quoted excerpt is witty and polemic, as is appropriate in a conference for shaking the minds of the audience; but it has also an enduring value insofar as it refers to one of the fundamental choices in computer design: the level of the working hardware. Sometimes it is referred to as

software hardware tradeoff. McKeeman then observes that the rudimentary philosophy of the computer (the Turing machine in his analogy) is not necessarily a negative aspect; special provisions may increase the efficiency in each class of problems. Nor should the complication of the machine programs constitute a handicap; compilers may present the users with a completely different machine. The real uneconomic aspect is probably in the redundancy of the problem descriptions; they require large memories and complicated manipulations.

Chu (1973) analyzes the "proximity" between the users' programming language and the actual machine language, for different computer architectures. Then he points out the advantages that can be obtained when the high-level programming language is the machine language of the computer.

A search in the literature has shown the following studies and implementations toward a higher working level of the hardware.

Anderson (1961) looks for an isomorphism between machine organization and the manner in which a user expresses a problem. He thinks that a good programming language properly reflects the concepts and abstractions of a particular class of problems, thus an efficient implementation of the language should constitute a good design goal for a computer. The language considered is ALGOL.

Mullery et al. (1963, 1964) aim to use a computer as directly as possible, without compilation. To this purpose, they develop first a general machine-independent high-level language, then a computer organization in accordance with that language. In recognition that data have always a structure, the language is strongly influenced by the choice of data format: a string of variable length, articulated up to eight levels in a tree fashion. The syntax of the language aims to be familiar by using nouns, adjectives, and verbs. It turns out that also the computer works in a phrase-structure mode. At the stage of the design, they find that the cost is slightly higher than that of an equivalent conventional computer. Reports on efficiency and practical use have not been found.

Melbourne and Pugmire (1966) implemented a subset of FORTRAN with a microprogramed compiler which works during the input procedure. Statements are converted into reverse Polish expressions, and identifiers are replaced by storage addresses. The resulting execution time is less than for con-

ventional compilers, and greater than for hand coding. They point out the ease of immediate interaction.

Bashkow et al. (1967) designed a partial implementation of a FORTRAN machine. Statements are interpreted in hardware; no provisions are taken for subroutines.

Weber (1967) implemented EULER in a conventional computer by microprogramming with additional ROM, partly as compilation, and partly as interpretation. The actual objective was to show that with the help of microprogramming existing computer hardware can be utilized to a higher degree than with conventional programming systems.

Iliffe (1969) built an experimental computer with a "base" more oriented to the elements of interest to the user. The basic type of elements are numeric (in different representations), control (corresponding to instruction sequences), and address (corresponding to information structures). With these elements, a process is constituted of few elements, typically less than a hundred. The details of these basic elements are not accessible directly. The functions of the machine apply to all three types of elements, with some restrictions. The syntax of the language is standard.

The hardware implementation of APL has been considered by Thurber and Myrna (1970). This work is oriented to future large-scale integration with which cellular arrays could be feasible. A matrix-oriented architecture is designed for which APL is an assembly language. Some of the APL instructions are microprogrammed, thus involving some sequentiality. A complete implementation of APL has been made by Hassitt et al. (1973) by means of microprogramming a conventional computer. Programs with only scalar operands run slower in this machine, and programs with large vector operations run faster than in the conventional approach.

SYMBOL is the latest experiment of a hardware-implemented high-level machine language (Chesley and Smith 1971; Smith et al. 1971). The language used "is fully in the mainstream of present languages". Practical evaluations are not yet available.

Vineberg and Avizienis (1972) have studied hardware solutions for taking advantage of computational independence within and among source statements.

Surprisingly, there appear to be no publications on hardware implementation of Decision Tables, a language that is particularly suitable for direct

implementation. Early computers were directly controlled by tables; analog computers, in a sense, follow decision tables. In modern times, Decision Tables implemented in software are recognized effective and are used (see McDaniel 1970; Low 1973).

All the studies and implementations mentioned in this section show particular advantages in relating more closely the structure of the programming language and the structure of the computer hardware. However, no significant impact was made on the mainstream of computers, in which language and hardware are developed independently. One can argue that the issue had not been attacked from its actual roots and with sufficient freedom. In the representation of Fig. 1, these studies and implementations can be visualized as attempts to bring C closer to B. The following comment can be made after the work described in this report: While attempts to exploit the capabilities of hardware have an unlimited potential, it appears not simple and easy to map directly the conventional hardware "space" with the "space" of phrase-structure language, as are all present programming languages.

1.4 A GLOBAL APPROACH

Current computer approach can be expressed as follows: The mechanization of the computer is assumed as given, then each problem is transformed (in fact, deformed) in such a way that it can be solved with that mechanization. In Section 1.2, one part of the typical transformations to which a problem is subjected was outlined. In Section 1.3, attempts were cited to make the computer mechanize higher levels of the problem representation.

An example has been shown (Schaffner, 1972a) in which the level of the computer mechanization has been raised to such an extent that we can think of an approach opposite, in a sense, to the current approach: Each problem is mechanized as the user sees it, then the computer is transformed, or synthesized, in such a way to match that mechanization. This approach sounds interesting, but the point is: Which is simpler, or more practical, the deformation of the problem or the synthesis of the computer ?

To find an answer, it is necessary to give a precise meaning to "mechanizing a process" and "synthesizing a computer". Thus, we need to analyze how human beings think of their problems, how thinking can be expressed, and in how many ways processing in general can be implemented by physical

devices. This brings us into the fields of psychology, linguistics, and mathematics. Moreover, in searching for suggestions on how to make up processing systems, it is appropriate to take an excursion into neurology.

Similar conclusions can be derived also from different viewpoints. For instance, from the concluding suggestion of the McKeeman address (see Section 1.3):

The obvious attack for programmers and hardware people together is to devise language that reflects what we want to do and how we do it, and machine structures effective in handling that language.

If we start from this attack with a fresh perspective, we have to recognize that programing in essence is a communication between a human being and a machine. Thus, we have to start from the disciplines pertinent to humans - psychology, linguistics, mathematics - and then look for all possible ways of implementing processes - engineering, neurology. Obviously, the consideration for user characteristics should not be limited to external aspects, such as studying the most relaxing color for the computer cabinets, or the least fatiguing type of display, but should go to the essential level of the nature of our thinking.

In conclusion, we are considering here a global approach of analyzing the nature of the processes as they are in the user's mind, the possible forms of representation, the possible ways of implementation, and then we search for solutions that might appear of interest, taking into account all the related factors. In an attempt to make this fresh perspective a useful contribution, we take here a position of audacious complete freedom from any constraint, or established interest, and observe as much as possible from a far distance, including collateral fields as much as possible. This interdisciplinary study seems to have brought interesting results: Common aspects in thinking, representations, and processing have been shown, for which a closer relation can be established among the three points of Fig. 1, suitable for practical application.

All these aspects have been studied in depth by different people, at different times, with different objectives. But it does not appear that such a global approach has been taken with sufficient commitment in regard to computers. Most work has concentrated on the phrase-structure languages, and computers have been kept around the notions of instruction-obeying processors and random-access storages.

Chapter 2

Why Abstract Machines ?

2.1 INFERENCES FROM PSYCHOLOGY

2.1.1 Introduction

As said in Section 1.4, we look for some insight into the processes as they are in the user's mind, for the purpose of improving the activities involved in the use of computers. This brings us into psychological considerations, which are discussed in this section. It is necessary to state that this section is not intended to be a survey of modern psychology; that the theories mentioned are as understood, interpreted, maybe biased, from our platform; and that our concern is only for picking out some clues useful, in an informative or in a heuristic sense, for our goal.

Two branches of psychology are particularly relevant to this study, the theory of thought and the theory of language. The history of these fields goes back to the classical philosophers, in their perpetual search for universals in nature and in human thought. As always, past patterns, experiments, and recurrences have a heuristic power, and may facilitate freedom from what may be merely incidental. A survey from a psychological viewpoint of the studies of thinking, from Aristotle's image to the present is given in Mandler (1964). Mathematical modeling is discussed in Miller (1964). A historical review of psychological studies of language can be found in Blumenthal (1970). However, only three references to the past are cited as an indication of the variety of relevant possible studies.

Modern graph theory confronts us with an impressive field impregnated with logico-mathematical properties. This fact clearly has deep roots in psychology, and can be found in a variety of forms at all times. A semi-historical survey of these reappearances is in Gardner (1958).

Axiomatic mechanization is one of the patterns of thinking. Association psychology applied it to modeling thought (see, for instance, Warren

1921). A mechanization, even if related to completely abstract objectives, is suitable for implementation by machines; a realizable abstract machine implementing this theory was indeed described (Reiss, 1962).

Gestalt psychology, as complement to the extreme schematization of associationism, reveals the existence of a natural "facility" in human thinking to proceed in terms of global entities.

In modern times a rigid concern for experimental evidence, together with more and more use of mathematical tools common to other disciplines, has brought psychology, in particular developmental psychology, to more fruitful connection with other independent fields such as mathematics and artificial intelligence. It is from developmental psychology that we will derive heuristic notions and support for certain assumptions to be made.

2.1.2 Developmental structures

Piaget describes three main stages in mental development (see Flavell 1963; Furth 1969).

1. Stage of sensori-motor operations (up to about 2 years of age). Co-ordination of perceptual and motor functions develops. The scheme of permanent objects progressively comes into being, in the sense that an object is known to exist even if it is out of the perceptual field. Elementary forms of symbolic behavior take place.
2. Stage of concrete operations (approximately 2-11 years). Properties of the present world are established (conception of space and time). Organization of complex operations develops, all directed to the concrete here and now. The world starts to be represented through the medium of symbols.
3. Stage of formal operations (from about 12-15 years). Hypothetical reasoning and deductive procedures become possible. Operations on internalized actions form new internal structures. With these means the continual growth of adult thinking proceeds.

A key point in Piaget's theory is that each new capability derives from integration and grouping of the capabilities previously developed. The new groupings result in new mental "structures". They are on the one hand more synthetic, organized at a higher level; and on the other hand richer in details, more differentiable ("structured whole", "set of all subsets", "ensemble des parties").

There is a continuity of development; the operations characteristic of each stage are based on operations characteristic of the previous stages. By grouping or integrating structures, new structures, and thus new functions, are produced. In turn, the integrating or grouping of these new structures produces still more new structures. Each class of structures constitutes at the same time the attainment of one stage and the starting point of the next stage. A structure at each given level is composed of structures of lower levels.

Piaget has related these structures to logic (Piaget 1950; Inhelder and Piaget 1958). Logic and psychology are two independent disciplines; the first is concerned with the formalization of internally consistent symbolic systems; the second deals with the mental structures that are actually found in all human beings, independent of formal training or the use of particular notational symbols, and regardless of consistency. Piaget applies the first as a theoretical tool in the description of the second. As an example, the new groupings that occur in the mental structures correspond indeed to the possible combinations in logic. He finds also that at different stages different logical functions can be performed.

In regard to the cause of development, Piaget invokes the mechanism of an equilibrium. The more unstable the system is in dealing with new situations, the more new integrations and groupings are induced. The broader the stability of a new structure, the more this structure becomes a permanent acquisition and is assimilated.

One interesting outcome in Piaget theory is that intellectual development occurs autonomously, independently of outside information, although very strongly affected indeed by it.

One fundamental concern for psychologists is cognition. Making a copy of objects pertaining to an objective world was one past theory; evoking innate images was another. For Piaget cognition is a process between perceptual data and operational structures. The structures adapt themselves, evolve in order to remain in equilibrium in the presence of new perceptual data. Perceptual data may or may not produce evolution depending on whether such equilibrium can be reached. Signals from the environment may remain meaningless and mute, or they may produce large consequences, according to the possibility of cognitive processes between the arriving signals

and the receiving structures (readiness of the receiving organism). Different structures will produce different outcomes. Thus cognition can be affected by working either on the input signals or on the internal structures. An object can be known only by being conceptualized to some degree. Knowledge is an adaptation and assimilation process.

It is interesting that here an input can not be "declared" to be something, from the outside; but it is what results from the interrelation between input and structure. This approach has to be taken because of the extreme complexity and variability of the structures.

From the theory of developmental structures we can derive heuristic notions. One can start from the available, or familiar, level, then proceed in integrating and grouping until a satisfactory solution is found. The familiarity, or the permanency of a level of integration is a function of the range of applications and success attained. We can also derive a heuristic notion of embedment of different levels. Or, put in a different way: rather than thinking of separated processes, one for each level (performed in sequence of time, or cascaded in space), we can think of a global, more developed, process (structure) comprised of all the levels.

2.1.3 The spatiotemporal frame

Piaget's modeling of mental processes emphasizes an evolutionary development of the mental structures during the different stages; in particular he refers to the "structural integration of concrete and formal operations" (Inhelder and Piaget 1958). Because the concrete operations are structured in a spatiotemporal frame, and because they are the basis of all the subsequent formal operations, we may deduce that all the mental activity of the adult, in its inner work, at some level, is structured in a spatiotemporal form. Of course, new, more effective structures can always develop.

Following a different course, other psychologists (Bruner et al. 1966) emphasize the evolutionary bounding up between perception and sensorimotor schemata (see Section 2.1.4). During the first months of life visual perception is inextricably associated with the handling and moving of objects. Because the structures thus learned remain fundamental throughout life, and are used also in later higher level intellectual activities, the frame formed at that time is embedded in abstract thinking.

In regard to the conscious level, people refer to "spatial intuition". In all abstract discussions, recourse is made to this spatial intuition; it is considered doubtful that even modern mathematics does not derive all information from spatial intuition (Beth/Piaget 1966). It is a fact that whenever an adult (even a logician) uses some kind of symbolic representation to describe a classification he is bound to think in spatial terms (Inhelder and Piaget 1958); Taxonomical trees and the Euler circles are examples. Even when dealing with the most abstract entities, we talk and think of mapping one "space" into another "space".

It is clearly meaningless to say that spatial relations are objectively determined from the environment. Experimental evidence shows that spatial notions do not derive directly from perception; on the contrary they imply a truly operational construction (Beth/Piaget 1966). Neither is there any usefulness in assuming that spatial categories are innate. The Piaget process of cognition shows that it is through the development of "intelligence", in reaction to stimulæ from the environment, that the individual constructs spatial notions. The objective constructions that we are accustomed to place within the environment are for Piaget identical with the structures of intelligence. In fact, everything we connect with objective, identity, causality, space, and time is regarded by Piaget as constructions and living operations (Piaget 1971). The fact that all human beings construct the same spatiotemporal frame is easily explained by common genetic structures. It is known that frogs have a visual and related processing apparatus quite different from ours (Lettvin et al 1959), and very likely their mental frame is different too.

In this light, it is probably fair to say that the spatiotemporal frame is basic to our thinking, and the frame remains, regardless of further complex mental structures that are developed subsequently. At the conscious level, however, one can "feel" different degrees of involvement of the spatiotemporal frame, because of strong internalization of new structures. From this the impression originates that "spatial intuition" may be only an heuristic device that is applied frequently. In sum, we may say that the spatiotemporal frame is the original substratum in which thought takes place; the degree in which this remains conscious varies in different cases.

2.1.3

If this is so, the hypothesis can be advanced that any mental structure, at any degree of symbolic abstraction, in which spatiotemporal form does not appear at the consciousness, can be restructured also in a spatiotemporal frame; and the new structure should appear at least very objective; whether it is also simpler needs to be seen.

An application of the above is in the following: If our objective is to develop ("to discover", in the sense that we do not yet have mental structures perform it) a logical system different from the familiar spatiotemporal system, obviously we have to reject (at the conscious level) the spatiotemporal frame; we have to work with the available structures to arrive at a consistent new system, and then we need to exercise the new system to internalize it. But if our objective is to describe a process that can be modeled well in the spatiotemporal frame, we create unnecessary work if we describe it in a different frame. This observation is made because in our opinion there is the following curious situation. The processes we give a computer can typically be well modeled in the spatiotemporal frame; all today's programs (a program is a description of a process) are expressed in the form of strings of symbols, form that needs a nonindifferent elaboration before it can be framed in our inner spatiotemporal structures - that is, be understood.

2.1.4 Words and imagery

In a broad sense, mental activity can be viewed as information processing, thus information has to be represented in some form, both internally and externally. In this fascinating, multifaceted, deeply intricate area, we will consider a few aspects very pertinent to our study. An extensive treatment of this subject can be found in Paivio (1971).

Bruner (Bruner et al 1966) delineates the successive development of three modes of representation. (1) Inactive (motor): at first the child knows his world by the habitual actions he uses for coping with it. Because these actions, or schema, are exclusively devoted to specific local goals, this system does not permit a real mental activity. (2) Ikonic (images): representation is made independent of action, and is permanent; it is formed in space and time, and allows for a degree of abstraction and anticipation. (3) Symbolic (verbal): the symbolic structuring of information permits the full development of intellectual activity.

The details and difficulties of such a classification are not of relevance here. Of interest instead is the notion that "each of the three modes of representation has its unique way of representing events. Each places a powerful impress on the mental life of human beings at different ages, and their interplay persists as one of the major features of adult intellectual life". We can also observe in the external human communication the use of a mixture of words, images, and gestures that are difficult to separate from each other.

Piaget (Piaget and Inhelder 1971), in reference to his developmental structures and cognitive processes (section 2.1.2), considers two main symbolic systems for representing information: the verbal system, and the system of mental images or imagery.

The verbal system derives from perception (typically, but not exclusively, from the audio apparatus), presents at the surface a serial nature, and has lexical, syntactic, and semantic structures. Its peculiar characteristics are the ability of categorization and a permanent precision. It is particularly effective for representing abstract concepts, and has a largely varying efficacy in representing the other object of thought.

The imagery system derives from perception (typically, but not exclusively, from the visual apparatus), through internalized imitation (in Piaget), or some other schematizing process, and presents at the surface a parallel nature. It evolves from static to kinetic to transformational nature, each form remaining available, and is capable of a continuous stylization from vivid views to pure abstractions. Its peculiar characteristics are the ability of condensing information, and an operational dynamics (as it will be elaborated below). It is very efficient for representing a global situation, and has a largely varying effectiveness in representing the different objects of thought.

The characteristics of the two systems are complementary and overlapping at the same time, and the two systems are used both in alternative and co-operative ways. For instance, we do not find the way to have a direct "mental image" of the concept of truth. On the other hand, we will never terminate in describing by words "all" the information we derive from a vivid image. Images (Piaget contests, *ibid.*) are not less symbolic than words; they are so in a similisensible form; in the schematization process that they generate, they retain the properties of interest and abandon the rest that

2.1.4

was perceived. They have the function to designate exactly as words have. They are very rich symbols; in some instances not very precise; at the extreme of stylization they tend to parallel words: a circle has the same categorization power as the word "circle".

Images have a peculiarly important role of interface between perception and the operations of intelligence. But the most interesting feature of imagery shown by Piaget (*ibid.*) is the intimate collaboration with the work of intelligence. The images help the evolution of the operational structures, and the operational structures help the evolution of images. The work of complex operations would not be possible without the features of the images, and the features of the images would not be possible without the collaboration of operations. (These psychological analyses are of tremendous heuristic value for our study, for they consolidate the notion of collaboration between data structures and operational structures.)

It should also be considered that imagery is the symbolic system by and large most available at the early stages of development, and thus the operational structures developed at that time are strongly oriented to images. Because these structures remain integrated into all subsequent structures, it is likely that imagery plays everywhere some inextricable role at least in some layer of all types of mental processes. It is a situation that goes with that of spatiotemporal frame.

The role of verbal language and of imagery in thought has produced in the past a sharp controversy. Associationists considered images as the material of thought - "thinking as a party of images". At the opposite extreme, behaviorists considered images as an auxiliary and occasional symbolism, and words the real content of thought.

From experimental evidence, psychologists today think that the mental processes exist independently of each one of the two representational systems. Neither one alone would fulfill the requirements for the mental activities of which man is capable; both are used in a complementary and cooperative way, together with other less prominent sign systems. As a consequence, these systems affect thinking. The language learned from the social environment indeed facilitates the development of thinking; and different languages may produce different mentalities (see Whorf 1965). On the other hand, it can be said that the autonomous (firm point in the Piaget theory) develop-

ment of formal operations implies a symbolic language of communication; if not thought, the individual spontaneously would generate some form of it. There is no doubt that images and words interact continually in every mental process. Which form is more dominant depends on the task, circumstances, and individual differences. Much has to be learned of both the verbal system and the imagery system; in particular, in regard to special roles of imagery (see Piaget). Everyone can observe in himself how one form can invoke instantaneously the other. Paivio (1971) views images and verbal processes as alternative coding systems, or mode of representation, and hypothesizes a double-coding system for memorization.

From an applicational viewpoint it is well known that "a picture is worth a thousand words"; but also "one word is worth a thousand pictures, if it contains the conceptual key". Thus, alternating the two forms permits better efficiency. Verbal symbols are exact schematizations very suitable for logic functions. But if a problem is given such as "Alice is taller than Mary; Elsie is shorter than Mary; is Elsie taller than Alice?" (in Bruner 1966, p. 9), translation into an up-down image allows a direct reading of the answer (Fig. 3 c). Mathematical expressions also become easier if they evoke some stylized ikonic or enactive element.

Moreover, imagery appears to have an unequalled power in its anticipatory capability in a spatiotemporal frame - the so-called geometrical intuition. Geometrical intuition can be considered as the practical counterpart of the Piaget theory of collaboration between imagery and operational structures. Note that this intuition can be trained to spaces completely different from those developed from perceptual activity. Paivio (1971), in critically analyzing the different viewpoints on the subject, suggests that imagery is the very basis of swift leaps of imagination in creative thinking. It is not by chance that the word "imaginative" has the meaning of creative.

It is important to make clear that imagery refers to the internal symbolic system used by the mental structures; it has nothing to do with the graphical means we use for communication with the environment, although, needless to say, one typically evokes the other. For instance, in the three-term problem cited above, we can represent (at the outside) the girls in different verbal and graphical forms, and evoke different mental structures.

In Fig. 3(a), the problem is represented in a verbal form, is modeled in a verbal structure, and it evokes mental verbal processes. Typically, a time of the order of ten seconds is necessary for the mental structures to solve the problem, when represented in this form.

In Fig. 3(b), the problem is represented in a graphical form, almost pictorial, but it is still modeled in a verbal structure, and evokes verbal mental processes. A time little longer than previously is typically used for solving the problem represented in this form, due to the extra coding of the girl's dress. This coding was necessary to avoid evoking imagery processes by showing different heights in the girl's portraits.

In Fig. 3(c), the problem is represented in a graphical form that is completely symbolic, rather than pictorial; it is modeled in a structural form, and evokes imagery mental processes. The observer doesn't feel like making any elaboration at all, and declares that the problem is "obvious". Note that much less symbols and means have been used in the representation (c) than in representations (a) and (b). The modeling has indeed a great influence.

In Fig. 3(d), the problem is represented with verbal means, but is modeled in a graphical form, and evokes imagery mental structures. As soon as the code of the problem is communicated, if not already guessed by intuition, the problem appears instantaneously solved, as for the representation (c).

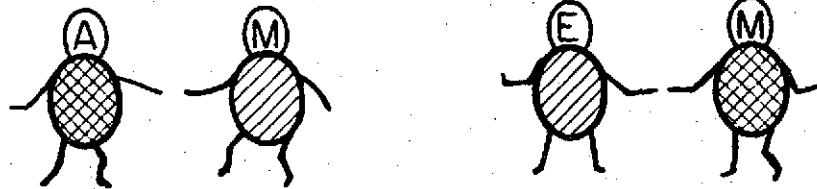
Clearly, it is not the use of external graphic means that is determinant, but the modeling of the problem in such a way that effective mental structures are evoked. Cognition is a process between external representations and mental operating structures.

At this point, it appears particularly remarkable that today's computer programs are developed in the form of word strings. Now we can attempt an interpretation for the fact observed in section 1.1 that, while most automobile users drive their automobiles themselves, a much smaller fraction of computer users write the programs themselves; this also considering other factors such as responsibility, cost, and intrinsic complexity. Driving an automobile is fundamentally a sensorimotor activity with full use of imagery; preparing a computer program in today's programming languages is a symbolic activity deprived of direct imagery.

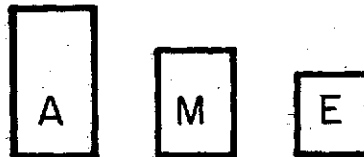
- Alice is taller than Mary
- Elsie is shorter than Mary
- is Elsie taller or shorter than Alice ?

(a)

A = Alice
M = Mary
E = Elsie



(b)



(c)

ALICE
MARY
ELSIE

(d)

Fig. 3 - Different forms of modeling and representation of a problem

We take the occasion here for an observation that will be of interest later. There is a well-known game called charades, in which a person or a group of people have to communicate to another group a message through mime, without any use of words. The message to be conveyed may consist of a name, an object, an event, or even an abstract concept. We see here a mode of representation that is not word structured, and that performs the same role as the verbal language. While on the one hand the rate of information transmission is typically very low, on the other hand certain nuances of feeling can be easily communicated by "body language" that would be difficult to express in sentences. The fact that this game is enjoyed as entertainment suggests that those activities related to creating and interpreting images and imitations do not constitute, subjectively, an effort.

Let us elaborate further. We can use mime to represent each single word of a message; and have observers reconstruct sentences from the mime activity. In this case we transmit through mime information that was already expressed in the form of sentences. We can observe that the rate of information transmission is low as is typical for mime, and that the capability of communicating feeling is poor, as is typical for spoken sentences. In this case, by cascading the representations we degrade the communication system - we sum the difficulties of the two forms of expression. Let us now consider two parties communicating by telephone (without picturephone); one party has expressed in geometric form certain properties, description of which he is unable to express in sentences; he transforms the geometric construction into sentences of a common language, and the other party reconstructs those elusive properties. In this case, the cascading of forms of representation does not result in a summation of difficulties. These exercises will be of interest for the discussions on page 170.

2.1.5 Concluding remarks

One of the most interesting aspects of psychology of relevance to our study is the consideration of experimental data on mental processes in terms of logico-mathematical structures. The subject obviously is a delicate one; always questionable is the value of a theory of one field applied to another field; but in the case of thought theory, the question is particularly incapable because thinking itself is involved in any human activity. The

independence of logic-mathematics from psychology is universally accepted, in the sense that no psychologism in mathematics, and no mathematics in psychology. However, from an epistemological viewpoint, in the sense of interpreting how science is the product of man's actual thought, a connection between the logico-mathematical structures and the subject's mental structures cannot be avoided. Fascinating in this respect are the Piaget's (1971) speculations for explaining the "mathematical necessity" in psychological and neurological terms. This necessity would be based on the need of equilibrium and closure of the mental activities in response to the various external stimuli and internal elaborations.

A further range of considerations arises when the peculiar features of computer programs are also involved, as it occurs in artificial intelligence (cf. Newell 1970; Papert 1973).

In our aim to gain an insight into point A of Fig. 1, we make treasure of the theories of mental processes. In particular, we acknowledge the belief that the role of the spatiotemporal frame is a primary, rather than a secondary support of thinking; the ascertained self-existence of thought regardless of each particular symbol system, such as verbal structures; and the prominent role of imagery among the symbolic systems. Furthermore, we feel a great heuristic power in Piaget's developmental structures, made of successive integrations and grouping of previously developed structures. We will resume these issues in section 2.6.

2.2 MODELING AND REPRESENTATION

2.2.1 In the previous section, we saw psychologists trying to model human mental activities. In this section we will survey how those internal mental activities try to model processes of the "external" world. The term external is in quotation marks to remind us that any object of mental activity (as opposed to the mental activity itself) is considered external even if that object is an internal creation without any apparent reference to an outside world, as it can be a problem of pure mathematics. Of course, thought can also make a model of itself; it is what we were discussing in section 2.1.

From an epistemological viewpoint, as Minsky (1965) says, "to an observer O^* , and object A is a model of an object A^* to the extent that O^* can use A to answer questions that interest him about A^* ". But here we are not concerned about the connections between A^* and A, e.g. the connections

between a physical experiment and the idea that the experimenter has of it; we are interested in the transformation of A into B as put in Fig. 1. Therefore epistemological questions are beyond the scope of our discourse. In section 2.1 we saw interesting facts and theories related to A; in this section we are going to see which general types of B have been used in the different fields; and in section 2.6 we will sketch the character of the A to B transformation that we propose for our goal.

2.2.2 A modeling activity, in the sense delimited in the previous paragraph, can have several goals, and corresponding viewpoints. It can have a heuristic purpose for arriving at the formation of a theory. It can have the purpose of filtering the information we have on a complex situation, for arriving at a simplified description that keeps the properties of interest and abandons those characteristics presently not relevant. It can be directed to the establishment of a procedure for obtaining a given or desired process. It can serve as a form of representation of a system. It can be used for changing the means with which a system or a process is represented.

Our goal, the representation of what the user has in mind, has connections with those viewpoints that increase in importance in the order in which they were listed; the discussion will be weighed accordingly. We see that modeling and representation are here strongly related. The choice of the form of representation determines the selection of the type of model; and the representation that is appropriate is the consequence of the modeling. Often a truly "problem solving" activity is involved, but we do not discuss directly this viewpoint (see for instance Kleinmuntz 1966); we will give it consideration indirectly by searching for tools that also facilitate solution finding. The main focus in this section is on the interplay of problems, tools, and people, as Naur (1965) put it; problems can be expressed only in terms of some modeling tools, different tools make different problems, and problems and tools exist insofar as they are recognized by people. Following the discussion in section 2.1, we can substitute people with the mental structures that people are capable of, or prefer to use.

2.2.3 From a historical viewpoint we can trace different forms that people used for modeling/representing systems.

Archimedes' "give me a lever and a fulcrum, and I will move the earth" is a startling example of a process modeled in the form of an abstract machine, and represented in a verbal form. It is conceived through the geometrical intuition and is presented in axiomatic words.

The pages of Leonardo's Codice Atlantico are covered with representations particularly interesting for the present discussion. They are mental conceptions, guided by mathematical knowledge, oriented to the attainment of specific tasks; they are not copies of memorized perceptions. Thus, it is hard to distinguish whether they are abstract automata or visualizations of real or possible objects. The form of representation consists of sketches where the imitation of movement is among the means of communication, and their spreading in the sheets makes an integral discourse with the words. There is a full use of imagery, anticipation, and verbal language.

The typical books of engineering in the last century are full of "artist's views" of machinery, visual and abstract at the same time, complemented with symbolic signs and words. As a result, they are functional and structural descriptions simultaneously. The practical success of this type of representation during the industrial revolution resides in the powerful capabilities of the geometrical intuition of the reader.

2.2.4 For heuristic purposes, let us examine modeling in a different environment: that of exhibitions; for example, world fairs, which are the occasions where the maximum effort is made to represent something. The visitor's first approach is an architecture, sophisticated, audacious, possibly symbolizing important aspects of the whole. This reminds us that it is psychologically pleasing, or effective, or both, to see at first an overall structure, and to have an initial preparation of the general spirit of the world to be seen. (Nothing of the sort is possible with present programming languages, which indeed have the task of representing the different worlds that programs are). Then, each issue exhibited in the pavilion is modeled and represented in the most appropriate form among the several available. Usually it is a mixture of graphic means and sentences; often actual mechanical static or working models are used. We have here a well-facilitated cognition process in the Piaget sense. The signals from outside have multiple structures in order to produce an effective communication with the variety of mental structures that the visitors may have. The success of an exhibition is indeed affected by the broad bandwidth of the language used. (Again the contrast emerges with the monocromaticity of present programming languages).

2.2.5 In a mathematical context, a model is constituted by a class of undefined mathematical objects, and relations among these objects. Modeling is to express in these terms a particular behavior or a specific structure. Mathematical models are the fundamental tools of all scientific work, and no general discussion of them is needed here. From a psychological viewpoint, they constitute one mode of thought processes, the one that has the well-known characteristics of rigor, stability, and deductive power.

A type of model of special interest is abstract algebra with related abstract spaces. It involves the notion of mapping an object of a class (or space) into an object of another class (or space). If the mapping is between two different models, a correspondence is established between the two. A model M' is said to be a homomorphic image of a model M when there is a unique mapping of the objects of M into objects of M' , and for every operation in question in M there is a corresponding operation in M' . The two models are said to be isomorphic, with respect to the operations in question, if the same conditions also hold for the mapping of M' into M .

A mathematical tool is developed and receives spreading usage insofar as it allows one to model with elegance one class, or several classes of problems. The greater its success in those classes, the stronger is the natural tendency to use it in other classes of problems. When a model reaches an almost universal application it produces the well known psychological effect that people tend to think that "reality" has the frame of that model. This is a very comfortable and practical situation, until some recalcitrant problem requires the introduction of a new, initially "odd", model. In psychological terms, these facts can be interpreted as the development and permanency of mental structures derived from cognition processes originated both from perception and elaboration of internal symbolic systems.

An interesting case is the following. The power of generalization of abstract algebra is well apparent. In the spirit of this model, a programming language has been attempted for nonnumerical processes (CODASYL 1962). The fact that difficulties have been encountered suggests that this type of model, in its pure form, while very elegant in a mathematical context, does not match well the common mental structures that are involved in specific practical applications.

2.2.6 In system theory, we see three classical approaches: input output response, state variable characterization, and the probabilistic approach. The last one is peculiar to certain situations in which an incomplete knowledge of the system is involved, and it is not relevant to the present discussion. The other two are both applicable to the same systems (see Dertouzos et al. 1972). For the first, the main tool is the calculus and the most interesting feature the feedback; the second is particularly suitable for discrete systems and more detailed discussion will be in section 2.3. The two approaches are obviously derived from "intuitive" mechanization of physical systems, and given their well-known success in varieties of applications, both should be available, in some form, to an A-to-B transformation.

More recently, complex systems are modeled in the form of computer programs. Since early time (see Minsky 1962), it was recognized that computer programs have some modeling power not easily achievable in the other approaches. Here the tool is a programming language. And indeed a programming language may allow of all the other forms of modeling. There is here a clear example of modeling of a modeling. A program can judiciously handle different modelings; there is a strong similarity with thinking; thus computer programs appear to be the most general and powerful tool for modeling. For this reason they are the main tool used in artificial intelligence (see for instance Barnerji and Mesarovic 1970; Vinograd 1973).

But this power of computer programs is obtained by simulation with a given machine, or programming language. And here we come to the object of our study, which can be expressed in the following form: how much can a machine or a language make computer programs replicate our actual thinking, rather than tortuously and inadequately simulate it ?

2.2.7 We noted already the ancient use of graphic means and their connection with the imagery of the mental processes. In recent years, graph theory had an impressive surge, both as an independent mathematical discipline (cf. Berge 1968; Golomb 1970) and as an interdisciplinary connective, especially with the computer field (cf. Read 1972), ranging from modeling the behavior of hardware (cf. Holt 1971), to modeling program structures (cf. Slutz 1968).

As usual, a programming language has been developed with particular features that facilitate the study of graphs (King 1972). But it seems clear that graphs have a potential for being themselves a programming language, or a part of it. A start in this direction can be seen in Rosenstiehl et al (1972).

We see graphs deserving a fundamental role in a general programming language for the following reasons:

- (1) They are symbolic, in the sense that they have the same power of categorization, the same precision and permanency as verbal symbols.
- (2) They are complementary to verbal symbols in that they are particularly effective in representing situations that are not effectively representable through verbal systems.
- (3) They can be functional and structural at the same time: a graphic means, while symbolizing a function, can also convey information of a structure or path, in time or space, related to the implementation of that function.
- (4) They constitute effective representations. Graphs have a wealth of properties; if these properties are made to correspond to the properties of a programming language, a very effective representation is achieved.
- (5) As a language, they evoke the imagery system of the user, with all the related heuristic power. The geometrical intuition of the programmer will very likely help him to find a solution for modeling his ideas, undoubtedly will guide him in the construction of the program, and finally will give him a facilitation for checking the correctness of the construction.

2.3 AUTOMATA THEORY

2.3.1 Introduction

Automata theory can be considered the branch of system theory that deals with the dynamic behavior of discrete parameter systems. Because discretization of parameters is an extremely practical schematization, and dynamic behavior is the most revealing insight we can have in a variety of systems of completely different natures, automata theory is growing autonomously, with inextricable connections with many other disciplines, and without a possibility to define convincing boundaries. For these reasons, a collection of different aspects of automata theory is discussed in this section.

There is no question that many notions peculiar to automata theory have been used, in one form or in another, in past speculations and practical applications. An example available in the literature is the work of Kutti (1928) for extracting and representing the essential behavior of complex telephone systems, regardless of the actual devices used and contingent details. However, the present comprehensive viewpoints of automata theory, which joins mathematical rigor with a satisfactory intuition, appear as a recent acquisition. The starting of automata theory can be placed in the Turing's (1936) introduction of a "machine" (abstract) as a constituent of mathematics. The only possible way to ascertain the computability of a mathematical function was found in the behavior of a machine. Undoubtedly, that event enlarged the context of mathematics.

In a completely different field, neurology, the need arose for formal modeling (see section 2.5.4). The growing field of digital circuits was demanding a general characterization of dynamical behavior (Huffman 1954). Then, automata theory became a recognized field with the publication of "Automata Studies" (Shannon and McCarthy 1956). At that time, mathematics, system theory, artificial intelligence, neurology, and formal expressions were already reciprocally involved. Moore (1956) gave the first explicit presentation of an automaton as a formal structure for modeling devices. In reference to the subject of the present report, this three-page paper appears to

be a beautiful example of collaboration of psychological fertility of the human mind (the development of a new mental approach), of mathematical rigor (the logical features of the model), and of adherence to the real world (the "unknown devices").

Mealy (1955) showed the use of the transition function for structuring the output. Wang (1957) started the increase of efficiency of Turing machines. Hopcroft and Ulman (1967) showed a way of generalizing automata. The connection with language was discovered (Chomsky 1957). The properties of finite-state machines were soon organized (cf. Gill 1962). The algebraic aspect were brought in light (Hartmanis and Stearns 1966). Most use of automata developed in mathematics (cf. Minsky 1967), in digital design (cf. Hennie 1968), in formal languages (cf. Luce et al. 1963; Ginsbury 1966), and in neurology (cf. Leibovic 1969). Comprehensive treatments can be found in Booth (1967), and Arbib (1969), among several others. The field has such fertility that several symposia and numerous papers and books are produced every year in connection with notions of automata.

2.3.2 Function and structure

One of the peculiar characteristics of automata is their dual aspect: functional and structural. For every "automaton" formally conceived in some mathematical form, it is possible to devise some operational structure (abstract) that implements an identical behavior. For every "automaton" conceived as an operational structure, it is possible to devise a formal expression that produces an identical behavior. The pioneering work of McCulloch and Pitts on this issue, in a neurological context, will be discussed separately in section 2.5.4. Each author that treats automata theory gives his form of demonstration of this correspondence. Kobrinskii and Trakhtenbrot (1965) give particular consideration to the issue.

Mathematical modeling and structural modeling appear to be equivalent. A mathematical function and its corresponding Turing machine are indistinguishable, or, more properly, they are two different representations of the same "object"; and for the moment we do not discuss if we might have a more essential insight of it. Therefore, we consider it meaningless to inquire whether an "automaton" is a mathematical object that can be simulated by an operating structure, or an operating structure that can be represented by a formal expression; an automaton is simply a notion that has those two aspects.

This property of automata has two important applications: (1) The synthesis of digital machines for performing given functions expressed in some symbolic form; and (2) The characterization by means of formal expressions of the functioning of given machines. Several procedures are available for obtaining these transformations in several classes of functions and devices (cf. Kobrinskii and Trakhtenbrot 1965; Harrison 1965). Obviously, there are several solutions, depending on the type of operational elements and type of structures used. Note that here there is a recursive procedure; each element considered can be in turn treated either as a new formal expression or as a further structure of other elements. In order for the system to be universal, there is a minimum operational complexity of the elements, for each class of structures and devices such as Turing machines, cellular automata, and digital circuits.

2.3.3 Formalizations

The dual aspect of automata, functional and structural, discussed in the previous paragraphs implies the treatment of the several "machines" by purely formal means. A variety of different mathematical models have been formalized (cf. Ginsburg 1962; Nelson 1968).

An example of such formalizations is the definition of a finite-state machine as a quintuplet $[S, I, O, \sigma, \lambda]$, where S is a finite set of state, I a finite set of input symbols, O a finite set of output symbols, $\sigma : S \times I \rightarrow S$ the transition function that maps the present state and input symbol into the next state, and $\lambda : S \times I \rightarrow O$ the output function that maps the present state and input into an output symbol. The mappings σ and λ are given in some symbolic form, or simply as a table. Often a graph representation is of great help in visualizing these functions. It is interesting to note that even in these purely symbolic treatments it appears again, as most effective, a representation in the form of a symbiosis of graphic means and verbal symbols that we saw preferred in different human activities and different epochs.

Very interesting relationships have been shown (Hartmanis and Stearns 1966) between abstract algebra and structural realizations of automata. The techniques of algebra suggest a great potential for characterization, composition, and decomposition. However, it does not seem that algebra can easily tell all the story of automata. The following comment is interesting

(ibid. p. 18): "We have found that it is often useful to think in terms of a little machine chugging away from state to state rather than in terms of abstract sets and mapping. For this reason, we now incorporate this view into our formalism."

The notion of automaton seems to be able to capture what is essential, common in the various mathematical or structural formulations. It would therefore appear the best vehicle for solving the question, as Arbib (1968) put it: given a problem, what is the computing structure best suited to it? Von Neumann (1948, 1958, 1966) was envisioning a general theory of automata, as a new formal tool, that could serve computers, neural systems, and dynamic systems.

2.3.4 Automata and languages

Automata can be seen as acceptors of input sequences that recognize one sequence from others by going into particular states, or producing a particular output. They can be seen also as generators of output sequences, under different input conditions. Input and output sequences are strings of symbols. Thus automata can be seen as recognizers or producers of symbol strings, a fact that associates automata with formal languages. Because languages are also defined by grammar, a correspondence between automata and grammar results.

These relationships had given automata a new facet, and opened a new way for modeling languages. The approach appears completely general; to finite-state automata, push-down automata, linear-bounded automata, and Turing machines correspond finite-state languages, context-free languages, context-sensitive languages, and recursively enumerable sets, respectively.

To the extent that a formal language corresponds to a computer programming-language, the string of symbols produced by an automaton can constitute a program (i.e., the string is interpretable as describing an automaton, the process in question).

Insofar as a formal grammar can be a schematization of a natural grammar, an automaton can be an element in modeling natural languages, but its productions are well far away from natural utterances.

2.3.5 Connection with psychology

The notion of automaton can be concretized in many other different contexts. Just to summarize, automata can be considered as: functions,

algebraic systems, logic systems, constructs, dynamic systems, sequential machines, logical networks, languages, expressions, programs, or graphs. It is often said that automata have several facets. A characterization capable to be consistent with all those facets might be: a system of mechanizations.

We can note that different classes of automata permit different types of behaviors, computations, processes, operations. Each class of automata is derived by integration and grouping of automata of other classes. Analogies with Piaget's developmental structures discussed in section 2.1 are striking.

Given the variety of contexts in which intellectual activity arrives to the consideration of automata, we conjecture the following interpretation. What are called "automata" (with all the difficulties to establish a general definition) are the external correspondences of mental structures that we use in varieties of mental processes; structures that are more involved in thinking than those corresponding to each single mathematical model; structures that are deeply imbedded with the system of imagery, and perhaps supported also by sensorimotor substructures.

Rather than viewing automata as multifaceted objects, we may more appropriately regard them as general purpose tools of our mental processes used in many classes of situations. Can a system have a model of itself? Von Neumann (1966) showed a procedure in self-constructing automata; Minsky (1965) elaborates on the issue. The dual aspect of automata is obviously due to the characteristics of our thinking; very likely the same characteristics that brought Piaget to hypothesize the two symbolic systems, the verbal system and imagery.

If this is so, automata appear the best candidates for constituting the framework of a general-purpose computer programming-language. The user models in this frame the various information he has in mind about the intended process, and then, expresses the model directly in that form through some means(*).

(*)Note that in conventional programming, a double modeling occurs. The process is first understood, mechanized (modeled) in the user's mind in some form (a first automaton); then, that model is remodeled in terms of a given programming language (an automaton to produce a second automaton); and finally, the user expresses the transformed model (the second automaton) through some means of verbal structure. In our approach, that will be elaborated later, we imply an automaton that is the user's original conception of the intended process, and means that allow him to express it directly.

The power of the automata framework is in its completely symbolic nature, the symbols (input, output, states) may correspond, as in thought, to simple elements or to other complex structures. Moreover, if an automaton is produced from the programming activity, a structure for implementation is readily available, because of the inherent structural aspect of automata.

2.3.6 Interchangeability between program and machine

Burks (1963) shows that a Turing machine $M * T = M * P \hat{\Lambda}$ composed of a finite automaton M , and a tape T comprising a finite program P and a blank unlimited portion Λ , can be substituted by a new Turing machine $M_1 * \Lambda$ composed of a more complex finite automaton M_1 and a blank tape Λ . Thus, all the computable numbers can be computed by a machine without program, but with appropriate finite automaton. In essence he points out two ways for obtaining universal computation: by varying the program in a fixed automaton, and by varying the automaton with, so to speak, a fixed blank program.

Then he recalls that any finite automaton M_1 can be substituted with a description $\mathcal{D}(M_1)$ on the tape, such that a Turing machine $M_u * \mathcal{D}(M_1) \hat{\Lambda}$ composed of a finite universal automaton M_u , the description $\mathcal{D}(M_1)$, and a blank tape Λ , can simulate the behavior of the machine $M_1 * \Lambda$.

In these terms, then, he discusses the interchangeability between active information (the description of M_1), and passive information (the description of the program P). Then, from this theoretical general frame, he points out that there are potential advantages in using active information more extensively than the passive one, and made the following suggestion.

5. A Machine Design Language and Automatic Programming

The idea of an automatic programming language is a commonplace now and it is customary to teach this language to the user of a machine, rather than the machine language. As noted earlier, an automatic programming language is the machine language of a hypothetical programmer's machine M_p with a certain organization, and this organization is presupposed in the automatic programming language. This suggests that it would be better to teach the potential user of a machine about the hypothetical machine M_p in conjunction with its language rather than to teach the automatic programming language in isolation from this hypothetical machine.

But what I wish to propose goes further than this. The hypothetical machine M_p was designed to solve all problems of a very wide class, and hence does not take advantage of the special properties of a particular problem. This limitation is inherent in the idea of a general-purpose computer. For many problems it is easier

to think of the problem in terms of a special-purpose computer especially designed to solve that problem. In doing this one will not be distorting his natural way of formulating a problem to adapt it to a particular computer. Instead, he can formulate the algorithm for solving his problem by designing a special-purpose computer analogous to the problem.

I suggest, then, that instead of always writing a program for a problem one should sometimes design a special-purpose computer for that problem. No doubt this suggestion seems preposterous. But the moral to be drawn from the work of Turing and von Neumann is that programs and computers are, to a large extent, interchangeable. Since this is so there cannot really be such a great difference between writing a (special-purpose) program and designing a special-purpose machine as it seems at first sight. There appears to be a great chasm between these two types of activities because the comparison between machine design and program writing is usually drawn between the long, involved design procedures which have produced our present general-purpose computers, and the relative ease of writing a program in a given rigorously formulated program language. But this contrast is not the relevant one here. The engineering design of an actual computer involves much more than the purely logical design of the computer, and this purely logical design is constrained by these engineering considerations. Moreover, in writing out or diagramming the logical design of a computer one does not have available a rigorously formulated design language comparable in power to the best current automatic programming languages.

Hence my proposal involves the development of a framework or language of great expressive power for specifying the logical structure of any computer. Experience in machine design and the use of flow charts for programming suggests that this language be diagrammatic as well as symbolic. Moreover, it is feasible to build a computer which can scan a two-dimensional diagram, so that the design of a machine in this language can be fed directly into the manufacturer's machine M_m . In other words, in designing a machine M one is writing $\mathcal{D}(M)$ in the proposed machine design language. The machine M_m must be told how to interpret the expressions written in the machine design language--this calls for an interpretive routine \mathcal{J} . To summarize, when one is interested in a computation $\eta(M^*D^{\wedge})$ he writes $\mathcal{D}(M)$ and gives it to the machine-program complex $M_m^*\mathcal{J}$. The number $\eta(M_m^*\mathcal{J}^{\wedge}\mathcal{D}(M)^{\wedge})$, which equals $\eta(M^*D^{\wedge})$, is then produced. (+)

Thus, my proposal involves, first, the development of a rigorously formulated machine design language, and second, the development of a routine for the automatic translation of expressions in that language into the machine language of the actual machine M_m . These two steps are, of course, the same as those required for the development of an automatic programming system $M_m^*\mathcal{D}(M_p)$: the machine language corresponding to the programmer's machine M_p must be worked out and the interpretive routine $\mathcal{D}(M_p)$ must be written. Likewise, the use of the automatic system $M_m^*\mathcal{J}$ is similar to the use of the automatic programming $M_m^*\mathcal{D}(M_p)$. In both cases one is given a problem. To solve the problem with $M_m^*\mathcal{J}$ he writes a description $\mathcal{D}(M)$ of a machine M which is equivalent to that problem. To solve the problem by means of $M_m^*\mathcal{D}(M_p)$ he writes a program P which is equivalent to that problem.

(+) \mathcal{D} symbolizes data.

The systems M_m^*J and $M_m^*D(M_p)$ operate on different levels of the hierarchy of Turing machines introduced earlier. It will be recalled that the universal machine M_u uses one block of input information to simulate Turing machines with blank (tapes, M_u^1 uses two blocks to simulate machines with one block of input information, M_u^2 uses three blocks to simulate machines with two blocks, etc., etc.). In this hierarchy M_m^*J is a case of M_u^1 , as is shown by the formulas

$$\eta(M_m^*J^D(M)^D\Lambda) = \eta(M^*D\Lambda)$$

$$\eta(M_u^1*J^D(M)^I\Lambda) = \eta(M^*I\Lambda)$$

and M_m used with $D(M_p)$ is a case of M_u^2 , as is shown by the formulas

$$\eta(M_m^*D(M_p)^P\Lambda) = \eta(M_p^*P\Lambda)$$

$$\eta(M_u^2*J^D(M)^I\Lambda) = \eta(M^*I\Lambda).$$

There are many possible approaches to our proposed machine design language. We will briefly indicate an approach which is suggested by von Neumann's cellular self-reproducing automaton but which diverges from it in a number of important respects. A finite of growing automaton of any power may be stipulated as the contents of a cell, provided that the specification of the automaton, either directly or via a chain of definitions, is reasonably simple. Thus one cell could store a number, with the understanding that the cell can store as many (finite) digits as the number has. For example, if it stores a ten-bit number 2 to begin with and is to store 2, 3, 4, . . . at various stages during the computation, the cell will automatically grow in size so as to accommodate the extra bits that are produced by successive multiplications.

In specifying a problem by means of a special-purpose computer one would assume as many serial stores, parallel memories, control units, etc. as was convenient. Data could be organized into blocks in natural ways. The control automata stipulated could direct operations like: sum the series in block A, monotone the data in blocks B and C, withdraw from memory all sequences having property ϕ , etc. There would be provision in the machine design language for defining new automata in terms of old ones, so once an automaton is specified others can easily be designed in terms of it.

Von Neumann has a fixed crystalline structure for his cells. We propose to allow new cells to spring up between old ones under the control of the computation. Suppose a list of words is stored in bins and at a later date new entries are to be inserted. This change would be conceived as an automatic process of inserting new storage bins between the old ones. This change must, of course, be accompanied by an appropriate change of the switches which connect these bins to the rest of the automaton. In general, storage and computing facilities would be created wherever needed and in a form suited to the problem being solved. Hence a batch of information would not be stored in a homogeneous memory, as is the case in current computers, but in a memory organized to reflect the organization of the information itself. That is, the memory would be divided into categories, subcategories, etc. in natural and useful ways, cross-switching connections would be assumed where needed, etc.

Current computers are organized into large, specialized units such as memories, arithmetic units, and controls. The reasons for this organization are to be found in the nature of the components from which computers are built. Since the special-purpose computers to be designed in our proposed machine design language are not to be built, there is no reason for organizing them in the conventional way. Rather, they should be organized in whatever way best accommodates the problem at hand. Consider, for example, a two-dimensional partial differential equation. It may be convenient to solve this equation by computing the value of a function at all grid points simultaneously, in which case the special-purpose computer should be organized to do this. It should be clear from the foregoing that in our proposed machine design language one could formulate machine organizations radically different from present ones.

In conclusion, let us review briefly how one would use the proposed machine design language. It would be most effective when applied to a problem capable of analog treatment, i.e., whose structure may be paralleled by the structure of a special-purpose computer which will solve the problem. In such a case the mathematical equation describes the behavior of a physical model. To specify the solution of this equation one describes in the machine design language a special-purpose computer which would operate analogously to the given physical model. The description of this special-purpose computer is supplied to a general-purpose computer which translates it into its own machine language and then solves the problem.

Whatever the practical feasibility of this proposed system, I think that the theoretical possibility of it illuminates Turing's and von Neumann's results on universal machines.

The results that will be shown in chapter 6 indicate that the suggested approach is not only of a theoretical interest, but seems to have also a tremendous practical potential.

2.4 CELLULAR SPACES AND COMPUTING STRUCTURES

2.4.1 The consideration of axiomatic, discrete spaces having certain properties (cellular spaces, or tessellation) provides us with another way to develop automata.

In a cellular space, each cell has a finite set of states, usually including a quiescent state; a transition function gives each cell the next state as a function of the present states in that neighborhood; a cell in the quiescent state, surrounded by cells in the quiescent state, should have the quiescent state as the next state. For reasons of simplicity, almost exclusively uniform and unlimited spaces, and deterministic functions have been considered. If a finite set of cells are initially set to a pattern of states, an automaton is created, and capabilities of computation and construction can be obtained.

2.4.1

Undoubtedly, this approach has been inspired by the biological structures. Its study is considered a part of automata theory; its implications extend into computing structures, chemistry, biology, genetics, and evolution.

2.4.2 Von Neumann did the first work in this field (von Neumann 1966; Burks 1970). He considered cells with 29 states and a transition function that depends on the state of the cell in question and on that of the four nearest neighbors. Actions can be produced by a cell to any of the four neighboring cells, in a synchronized time sequence.

In this substratum organs can be synthesized, such as pulsers, recognizers, transmission channels, wire crossings, tapes, constructing arms, etc. With these means, finite automata and Turing machines can be formed, and their work simulated. Thus universality in computation can be achieved.

But also construction can be obtained. Having set initially into the cellular substratum a finite configuration of states that forms a constructing automaton and that contains a plan (the description of another automaton), new automata can be constructed, in some other region of the substratum, and left to operate independently. An automaton can reproduce itself, i.e. produce another automaton that contains the same capability of construction. Universality in construction can be achieved. An automaton can also construct an automaton of higher complexity than itself. Obviously, such a mathematical finding has implication for biology.

2.4.3 Moore (1962) considered transition functions depending on the eight nearest neighbors. He showed the phenomenon of Gardens of Eden, configurations that can exist only at the origin.

Codd (1968) found universality of computation and construction with eight states and dependency on four neighbors. A two-state cellular space can have universality in computation and construction but not with dependency on only four neighbors.

Yamada and Amoroso (1969, 1971) introduce a general d -dimensional tessellation automaton defined as a quadruplet: the dimension d of the array of cells, a neighborhood index, a set of states, and a set of next-state functions. In essence, it is an infinite regular array of identical finite-state machines with a transition function that can change from step to step, and is

uniform for all the cells. Then they formalize a structural and behavioral iso- and homomorphism for this type of automata.

Grosky and Tsui (1973) expand further the analysis of tessellation automata by considering spatial non-uniformities. Moreover, they consider under a unified formalization both sequential and parallel transformations.

It is instructive to observe that in Turing machines data structures and operational structures are well distinguishable -- the tape and the finite-state part. In cellular automata instead, data and operational structures are inseparable; operational structures come out by structuring data in space, and data structures are the result of operational structures. Of course, in certain instances, a structure can be seen as stored data, or as an operating device. Cellular automata exist in an environment (substratum) made of the same elements as themselves; they differ from their environment only in that they are "organized", while the environment is not (Burks 1970).

In the last generalizations, tessellation automata, from spatiotemporal structures (as they were at the origin), become purely mathematical structures completely disconnected from any geometrical intuition of the conventional space.

2.4.4 Holland (1960, 1965) has formalized a class of substrata in which the single cell has storage and operational capacity. It is a homogeneous, discrete, unlimited space, thought as an array of modules, defined by a quintuplet

$$[A, A^{\circ}, X, f, P]$$

where A determines the geometry of the discrete space, e.g. the dimension; A° determines the standard neighborhood, or connectivity available to each module; X determines the storage capacity of the module; f determines the operational characteristics of the module; and P determines the path-building (addressing) capability of the module. A specific quintuplet characterizes a particular space. This class of spaces admits representatives structurally and behaviorally equivalent to Turing machines (with one or more tapes), von Neumann's cellular automata, logical nets, and potentially infinite automata.

In particular, Holland has considered spaces with a potential connectivity (construction of paths) by which a cell can affect another cell in one time step, regardless of the distance. Thus, this space is more efficient than the von Neumann tessellation. In this space, a finite automaton is

defined by a quintuplet

$$[I, S, O, f, u]$$

where I is a finite set of inputs (the ensembles of signals at the input lines), S a finite set of states, O a finite set of outputs (the ensembles of signals at the output lines), $f: I \times S \rightarrow S$, and $U: I \times S \rightarrow O$ the transition and output functions, respectively. He uses these means for studying adaptive systems. In one approach (Holland 1970), a hierarchical organization of schemes of blocks at different levels is used; at each level, a block is considered as a primitive, and can be substituted for a better one, provided that it satisfies the input-output interface. Initially, a set of automata acting as generators is given with the capability of measuring the success of operations within themselves in relation to the environment. Then the automata duplicate and generate new structures. The construction proceeds by grouping and integrating blocks at the different levels. (Similarities with theories in section 2.1 are remarkable.)

This approach is interesting also because it shows the possibility of describing automata implicitly. Rather than giving the description of all the possible occurrences in an automaton, a set of generating elements and relations on their production, or growth rules, is given. Typically, an implicit description is more compact than an explicit one.

On the same line, a computer structure capable of executing an arbitrary number of subprograms simultaneously also has been described (Holland 1959). It consists of a two dimensional grid of identical, synchronous processing modules. Each module has a storage register, a certain number of auxiliary registers, and associated circuitry. Each module can exchange data and action signals with its four nearest neighbors. The storage registers contain instructions or operands, the auxiliary registers contain the state of the module and other control information. At each time step (a cycle of the computer) there are three phases: (1) the modules may acquire new data from outside; (2) communication paths between modules are established (that may be also conservation of previous paths); and (3) the prescribed operations are executed. A module, from inactive, can become active, transfer the action to a neighbor, and become again inactive; the predecessor and successor are in space rather than in time. Several independent or interactive routine (corresponding to paths) can be executed simultaneously in the array. In a sense,

it is a spatial counterpart of a plurality of interconnected Turing machines with a much higher level of actions.

From an applicational viewpoint, this approach has a rich potential for very ingenious problem solutions. However, the structure seems too much predetermined although completely general; in certain cases it would appear a natural frame for the problem (e.g. two-dimensional numerical models), and in many others it would require an extensive remodeling of the problem on the part of the user or a compiler. In regard to the utilization of the modules (efficient use of the eventual hardware), this approach seems not favorable, unless a different level of automata theory, such as that needed for understanding biological processing systems, could give the structure a different power. However, the real difficulty does not seem to be in the number or in the utilization of the elements (in view of the present or future technology), but in the formidable task of programing (cf. Comfort 1962). To help this task, the concentration of arithmetic power in a few modules has been also considered (Comfort 1963).

2.4.5 More in the direction of macromodules is the SOLOMON computer (Slotnick et al. 1962). It is an array of identical processing elements working in parallel under the supervision of a central control unit, "in an arrangement that can simulate directly the problem being solved". The quoted sentence shows the main aspiration of this solution, which is clearly relevant to our goal. The work related to the SOLOMON computer has been a contribution to the ILIAC IV computer, but it did not produce a new line of general-purpose computers; we here discuss from our viewpoint the potential and limitation of this approach.

The problems to which SOLOMON was particularly addressed include parallel computation in mesh points; for these operations, the array of processors, capable of direct communication with the four nearest neighbors, and executing broadcasted instructions, constitutes a very efficient structure well matching the characteristic of the problem. But, at the same time, this pre-established configuration of the array causes a very low performance when the problem has a different structure. Because the processors are so numerous, they are inevitably simple for obvious reasons of cost; thus, when an operation would demand a sophisticated processor, either the execution is slow, or complicated programing is required for compensating by using several

processors. The memory is subdivided in sections attached to each processor; while this solves the problem of access to the many processors, in other respects there is less flexibility and utilization than with a single main memory. Input and output are provided at the edges of the array; for particular problems, this is a completely satisfactory situation, for most problems, it involves additional transfers and programing. In summary, the approach is valid for a particular class of problems, but it has not appeared as an efficient solution for a general-purpose use, in terms of both hardware and programing effort.

For the specific application of recognition of patterns in a surface, Unger (1958) had considered a processing structure composed of a two-dimensional array of modules that make bit by bit operations.

Recently, the interest for array structures has a revival because of possible application of their properties to integrated electronic substrata (see for example, Minnick 1967; Akers 1972; Jump and Fritsche 1972).

Further comments on cellular spaces will be made in section 3.4.3.

2.5 INFERENCES FROM NEUROLOGY

2.5.1 Introduction

The neural systems of living creatures are considered to be the organs mostly responsible for their sensorimotor processes and psychological activities. Thus it is plausible to expect a correspondence between these activities and neural structures. While for the vegetative functions and the elementary sensorimotor processes specific neural structures are clearly distinguishable that perform specific functions, the functioning and the structures of the neural system becomes more and more elusive when higher level activities are considered.

One would naturally think of an analogy with computers, where no resemblance can be found between the structures of the different processes performed and the structure of the hardware that performs them. It is only through the notions of coding and of hierarchical layers of virtual structures that the relation between structure and function can be reconstructed. However, there are fundamental differences between computers and neural systems. Neural systems have undoubtedly some probabilistic aspects. while

computers are completely deterministic in their functioning. Neural systems have the capability of structural self-organization, a capability unknown to computers. Moreover, the organization of neural systems appears completely different from that of computers. Thus the interest for our study to look into the structures and functioning of neural systems, especially for the high level activities.

It is assumed as a fact that psychological activities, regardless of their level, base their existence in the neurological substratum. However, the connection between the two is so poorly known that it cannot be analyzed directly. At present, psychologists look for understanding mental processes through psychological experiments and modeling theories; neurologists study structural and physiological aspects of the neural systems; and finally neuropsychologists attempt to make inferences between the findings of those two fields. Among the most interesting questions involved, beside the epistemological problems, there are the nature of cognition and knowledge, the cause of evolution, and the "necessity" of logicomathematical structures.

The notion of cognition as a process between a receiving structure and signals from the environment (see section 2.1.2) brings Piaget (1971) to consider knowledge not as an additive accumulation of experience, but as a steady improvement in cognitive instruments. As an example, the functioning of the cortex, which is hereditary insofar as the genetic substratum that permits its functioning is concerned (the development achieved by the different species is rather precisely determined), is completely evolutionary as cognitive structure. The acquisition of knowledge is viewed mainly as a development of structures.

As for the cause of evolution, after the two alternatives of Lamarckism (structural changes produced by changes in the environment) and of Darwinism (natural selection produced by the survival of the fittest), the hypothesis of autoregulation seems more satisfactory. It is a general autoregulation (loc. cit.) that occurs through cognitive processes between the organism, where the nervous system is the main organ involved, and the environment. It is an equilibration in the search for closure of the open system of the organism plus environment.

In regard to the "necessity" of logicomathematical structures, Piaget (loc. cit.) presents the fascinating hypothesis that this fact is a highly

2.5.1

differentiated extension of biological general autoregulation. Logicomathematical structures are frames that permit equilibrium and closure in their exchanges with the environment (from which the appearance of pertaining to a physical world), and in the reflective constructions of thought (from which their appearance of "necessity"). They do not exist per se, but emerge from the functioning of the mental structures, as soon as the functioning is used for solving problems (that is, reaching equilibrium and closure). Thus logicomathematical structures are endogenous developments. They are similar in the plurality of individuals because of a similar genetic substratum and similar stages of development. But because they are developments, these "universals" are also open to evolution. This view encompasses the previous philosophical views of innateness, and derivation from an outside world.

In this context, the question is not one of searching for a direct isomorphism between psychological and neural structures. Cerebral functioning should be seen as an expression of very generalized forms rather than of particularly organized forms at independent levels. As an example, neurons are capable of precise logic functions since birth, but the child is not; then, the child develops sensorimotor schemata that contain some relationships but which are still elementary by comparison with the logic of neurons. It is only after twelve to fifteen years that logic operations become possible for the individual, and they develop to a much higher complexity than that of the known neuron functioning. Moreover, there is the fact that on the one hand, entire classes of different physical structures can correspond to each function; and on the other hand, different functions can be performed by a given structure. Precisely because of all this, an examination of the neural system has a heuristic interest for the present study, particularly in reference to the central neural system, which is related to the higher level functions.

2.5.2 A look at the central nervous system

In mammals, the vegetative functions are regulated by neural systems located in the spinal cord, hypothalamus, etc., and constituted by well definable, specific structures. On the other hand, for providing the so-called higher functions, there is a sophisticated neural system, referred to as the central neural system (CNS) located in the cranial box, the brain, composed of structures with a high self-organizing capability. The CNS of man is

characterized by having a significant portion initially in an apparently uncommitted structure, which slowly self-organizes with the development of the individual.

Structurally, the human brain is about one pound of material composed of a large number of cells and fiber bundles. A few of these cells (however, about 30 billion), called neurons, have a well recognizable structure and have some aspect of their functioning known. The most numerous type of cells, called glial cells, have practically unknown roles in addition to providing a metabolic function between blood capillaries and neurons, and the myelination (isolation) function to the neuronic structures. The external shape of the brain, with its lobes, convolutions, and fissures, has a phylogenetic derivation, as it can be inferred by biogenetic law from the development of the embryo.

From the viewpoint of our study, there is a heuristic interest in the neurons, about which a degree of knowledge is available, in the functional structures of the cerebrum, the strategies used, and the interpretations and hypothesis that are given.

Neurons have several, but characteristic, shapes and sizes; they have typically several ramified branches, called dendrites, which can be thought of as inputs, and one, typically long, conduit, called the axon, that can be thought of as an output cable. Neurons have chemicoelectrical processes by which, in first approximation, they act as threshold devices; when certain input conditions are exceeded, an impulse is produced that propagates along the axon. The functional aspect emerges from interconnection; the inputs of each neuron are connected to outputs of other neurons, and some to sensory cells through chains of other neurons; the output of each neuron is connected to the inputs of many other neurons, and some to muscles through chains of other neurons.

The main peculiarity of the neural network seems to be in these connections, called synapses. They are points of contact between the axon (output) of a neuron and the dendrites (input) of other neurons. In few cases they seem to produce a direct electrical contact; as a rule, they are constituted by a thin - less than 1 micron - membrane at the contact of the dendrite and the axon, where small packets containing few thousands of molecules can be released. A synapse can have either an excitatory or an inhibitory effect.

"Nearly every synapse that has been carefully studied has been proven to have unique features" (Eccles). In the CNS a neuron has typically several tens of thousands of synapses. Clearly, such a substratum is suited for an extremely large variety of structures, for any degree of detailed variation, and it can easily cope with malfunctions of single elements. The complexity of functions that can be performed even by a small group of neurons in the central neural system can easily be inferred. In the human cortex alone, there are more than 10^{14} synapses.

Ever since Galvani's experiments with frogs, there has been assumed a connection between electricity and the nervous system; but it was not until the 1930's that Hodgkin was able to produce the first analysis of it. Also chemical transmission was first proved only in the 1920's by Loewi. A collection of the main contributions to cellular neurophysiology can be found in Cooke and Lipkin (1972). Studies on synapses can be found in the numerous publications of Eccles.

The cerebellum - A specific, region of the brain has recently been studied in its structural/functional aspect -- the cerebellum. It provides for the regulation of movement, autonomous and voluntary; it receives information from the higher centers of the brain (the cerebral cortex), and coordinates the muscular movements responsible for behavioral acts.

Structurally, the cerebellum has a layer (A in Fig. 4) of Purkinje neurons (labeled 1) very rich of dendritic branches (inputs), and a layer (B in Fig. 4) of granule neurons whose axons (outputs) extend in parallel fibers (labeled 7) running through the dendritic region of the Purkinje neurons. Moreover, scattered in those layers, there are three other types of neurons, the Golgi, the stellate, and the basket cells (labeled 2, 3, and 4, respectively) which synapse in the same network. Each Purkinje neuron has up to 200,000 synaptic contacts. The dendrites of the granular neurons synapse with one type of the arriving fibers, and the axons of the Purkinje neurons are the output fibers of the cerebral cortex. Another type of arriving fibers synapse directly with the Purkinje neurons.

The functioning of this extremely intricate network can be seen globally as dynamic patterns of inhibitions in a multitudinous bombardment of pulses, at specific regions of the cerebellum for the different parts of the body. These patterns are learned with exercises; at birth, relatively few synapses



Fig. 4 - Structure of the cerebellum in man

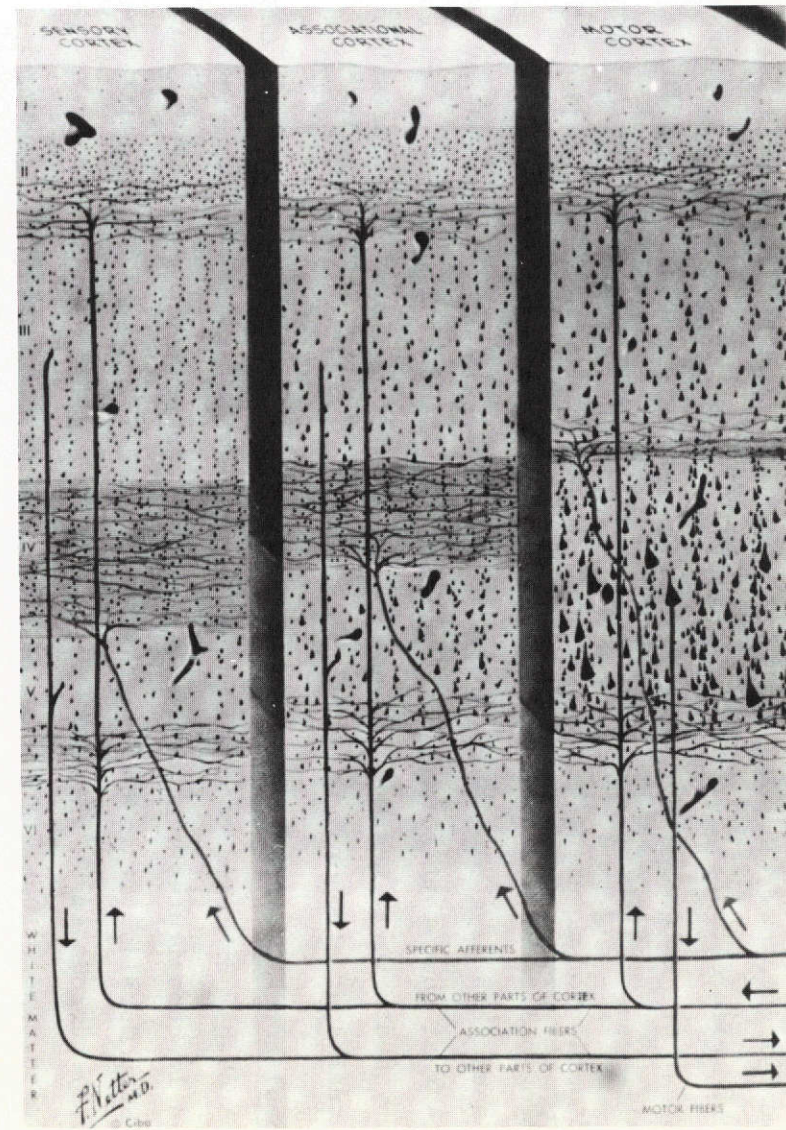


Fig. 5 - Structures of the cerebral cortex in man

I
II
III
IV
V
VI

2.5.2

exist in the dendritic region of the Purkinje neurons; repeated sequences of movement induce new synapses (Eccles's "spines") that produce a progressively more refined control of the movements. Each individual forms a personal endowment of "spines" that gives the individual motor ability and style.

It seems that here the spatiotemporal frame develops, from the lifelong coordination of movements toward goals. It would be of interest to see which neurological counterpart can be found for the role of spatiotemporal frames discussed in a psychological context in section 2.1.3. There are many experiments that show that formal thinking is accompanied by imperceptible muscular movements (e.g. eye movements), as if we were dealing with objects having an actual spatiotemporal structure.

Cerebral cortex - The cerebral cortex is a highly organized part of the CNS, that probably exhibits to the highest degree a self-organizing capability. In man it constitutes the largest portion of the brain.

Structurally, it is composed of neurons arranged in columns and layers (Fig. 5) surrounded by glial cells and blood vessels. Six layers are generally recognized, the fifth of which contains most of the pyramidal neurons, the type more characteristic of the cerebral cortex. The axons of these neurons constitute the output fibers, which go to other regions of the cortex and to the outside of it; the input fibers, from other regions of the cortex, split in branches between layers V and VI, and between the II and III layers. Other specific afferents, coming from outside the cortex, split into layer IV. The pyramidal neurons have ramified dendrites rich of spines (interpreted as acquired synapses) and are surrounded by axons of stellar neurons that produce an inhibitory feedback. The neurons of a same column seem related to the same elementary operation. This general structure is observed in all parts of the cortex, though with marked variations.

The complex connectivity of the cortex is not significantly deciphered. Luria (1966) sees the cortex as "an organ capable of forming functional organs"; functionally thought of as a result of comparing what was planned with what in fact takes place.

There are some areas of the cortex that appear committed to sensory and motor operations, with a mapping to the different parts of the body. Other regions, the largest ones in man, appear genetically uncommitted and spatially not strictly specific. In the uncommitted regions, an area has been

related to speech, usually in the left hemisphere, in some individuals in the right hemisphere. The homologous area in the opposite hemisphere has been related with the interpretation of spatial relationship and visual experiences (Penfield 1965). There has been always a search for mapping the various mental functions into areas of the cortex; however, this does not seem the appropriate approach.

While for sensory motor functions some definite neural structures are distinguishable and localization is possible, it seems impossible to do so for the so-called intellectual activities. More than a wired diagram is a question of coordination of involvement of many parts of the brain. In modern neuropsychology, the concept of function is not related to the properties of specialized neural regions and organs, but is understood as the product of a temporary collaboration of dynamic structures spread in the entire nervous system. The higher mental functions start as systems based on relatively elementary sensory motor processes; with development, they take other more direct restructuring, dropping many passages that were related to their formation. There, different neurological patterns (probably corresponding to different psychological strategies) may be found in different individuals.

Memory - The learning of a function - perceiving a situation and memorizing it - is the most impressive capability of neural systems. The human CNS has this capability developed to an astonishing degree; yet its neurological context has been the most elusive, both for the short term, and for the long term memory forms. Only tentative, partial hypotheses are available so far.

From the structural studies, the synaptic theory of learning and memory develops (cfr. Eccles, 1964). Functions and perceptions produce spatiotemporal patterns of impulses in the cortex neuronal network; repetition, or permanency of this pattern, produces a higher efficiency of the synapses involved; a stronger channeling of the pattern results, with the consequent formation of an "engram" of synapses. The fact that cuts and extirpation of any portion of the cortex do not destroy any specific memorization makes this theory unsatisfactory. The huge amount of information that can be retained in the permanent memory turns the theory to little use.

Genetic and immunological memories have been explained with molecular structures. Variations in RNA have been found in neuron and glial cells after learning activity. Thus a molecular theory of the psychological memory

2.5.2

has been suggested, based on some hypothetical coding system. This approach is compatible with the experiments of rats that acquire skill after injection of RNA from trained rats (cfr. Schmitt 1968), and worms exhibit a capability belonging to other worms they cannibalized (McConnell 1968). However, the theory, at the present stage, does not explain how the process of memorization occurs.

Widespread electrical activity during learning periods, revealed by electroencephalograms, has suggested memorization processes in which neurons, glials, and intercell liquid are involved (Adey, 1968). The fact that memorization appears diffused in all the cerebral volume, and any new information matching some part of the memorized content is able to recall the entire content has suggested an analogy with holograms and holophones (Longuett-Higgins 1969).

Modeling - Given the structural and functional complexity of the CNS, some kind of simplified models are a necessity for analyzing and understanding its performance. But, as one can expect, models of the neurological counterpart of the psychological activities are extremely difficult and limited.

The all-or-none response of neurons and their concatenations suggested McCulloch and Pitts (1943) to model the behavior of neural structures by means of networks of formal neurons. This approach will be discussed separately in section 2.5.4, because of certain theoretical implications of particular interest for the subject of this report. A formal network, as are other types of automata, is suitable of a variety of treatments; for instance, as an algebraic structure; thus they give the possibility, at least theoretically, of a logicomathematical treatment of neural systems, starting from their structural aspect.

The multitude of structures that are discernible in neural systems has suggested a genotypic modeling (Rosenblatt 1962) in which sets of rules for generating classes of systems are used, and then the results are compared with the psychological behavior. Rigorous analyses have been made of possible parallel structures that can be applied to the modeling of perception (Minsky and Papert 1969). One interesting feature of this approach is its learning capability (the perceptron convergence theorem) by means of a small, intelligent kind of memorization.

The extremely large variety of dynamic behaviours that are possible in

arrays of active elements has promoted several attempts to model in such terms the functioning of neural systems. The reverberations of Caianello (1961) and the resonances of Reiss (1964) are examples.

After the initial formalization of McCulloch and Pitts, automata theory always appeared an interesting candidate for modeling neural behavior. Its main power, with respect for instance to stimulus-response theories, is in the elegant treatment of past histories of the organism by means of the notion of state. Von Neumann (1958), in comparing the characteristics of biological systems and of man-made processing systems, was envisioning a general theory of automata that could be a theoretical basis for both. Arbib (1968) pointed out that the potential of automata theory lies not so much in the modeling of actual neural structures, but in modeling the different aspects of the information processing involved in neural systems. Rosen (1969), in turn, suggests to use the automata-theoretic description independently for each of the several hierarchical levels that can be defined in the functioning of the neural systems; in this case, the interpretation of state will be different at different levels.

Cybernetic models seem very appealing in the light of the autoregulation aspects inherent in nervous systems; however, they do not lead to detailed correspondences with the actual neural structures.

2.5.3 Inferences from the central neural system

(1) Given a certain similarity of tasks between the central neural system and computers, we shall try in this section to derive some heuristic inferences from the former that might be of interest for the latter. Obviously, the solutions adopted by biological systems are not necessarily the appropriate ones for man-made systems. Very likely, they are not, for several reasons. Neural systems are as they are because of an evolutionary development that happened in the epithelial system from which they derive; computer design can start from any approach (man-made high-speed vehicles use advantageously wheels and roads, while biological runners are bound to use legs). The CNS should work in the worst conditions, for instance, when an animal is jumping in a harsh forest escaping from a deadly danger; computers typically are protected, often being kept in air-conditioned rooms. Biological systems are made of unreliable but self-repairing components; man-made systems are made of reliable but fixed components. However, given the capa-

2.5.3

bilities of that one-pound system that is the brain, we can suppose that there are certain directions of some kind of mathematical essentiality; thus we may learn something or at least have some heuristic suggestions, that might be useful also for computers.

(2) A characteristic approach in the CNS is the self-organization. New functions are automatically produced, under compulsion of the environment, from presently available functions. Note that the brain does not carry a self-construction; the number of neurons decreases during life. So far as neurologists know, it is the connectivity that is developed. If structural changes are indeed implemented in neural systems based on chemical-biological processes, which require time, structural changes might be even more suitable in man-made technologies, which usually have much faster response. Particularly challenging is the capability that seems present in the CNS to restructure automatically the substratum for performing new functions more effectively and directly, as a consequence of repeated use and time. Self-organization is being tackled now in artificial intelligence, in the form of programs; but it is completely extraneous to the conventional use of today's computers. We can see an aspect of self-organization in the implicit description of automata mentioned in section 2.4.4.

(3) The CNS is viewed not as a collection of independent, specialized resources (as is the rule in present-day computer analysis), but as a collection of different rearrangements of the available substratum, toward the goal of the moment. I feel rich of heuristic suggestion a neuropsychologist view "that the material basis of the high nervous processes is the brain as a whole, but that the brain is a highly differentiated system whose parts are responsible for different aspects of the united whole" (Luria 1966, p.35). This approach offers an alternative to the one used in today's computers: that of concurrent work of different parts, with the related task of searching for possible parallelism in the processes. The alternative is concurrency of the different branches of the substratum into a coordinated work for the task of the moment, as in our thinking in which all the mental faculties are reorganized each time for the present task. Obviously, if parallelism is inherent in the structure of the process, that parallel structure will be assumed by the substratum. The brain strategy seems to be far from both the specialized resources of conventional computers and from the uniformly dis-

tributed potential of the cellular spaces; it seems to use the integration of differentiated substrata, and the systemistic collaboration of already developed structures.

(4) One of the most peculiar characteristics of the neural systems is the inseparability of data and functions; we can refer only to structures that account for both. We have already noted this characteristic as peculiar to automata, in particular of those realized in cellular spaces. This situation is remarkably different from that of today's computers, in which data and instructions are two disjointed entities.

(5) Memory is one of the most impressive capabilities of the CNS. Yet, neurologists have been unable to find any physiological behavior that can be interpreted as an addressing function. In psychological context, the conscious recall of past memories seems related to association; the subconscious utilization of past information has an unknown mechanism; vivid revival of forgotten episodes occurs also by electric stimulation of the uncommitted cortex. In some cases, it seems evident that information resides in some form in the processing structure itself, rather than in a "memory" from which it is retrieved by some means. In general, it appears that data and operational structures are embedded in each other. From a neurological viewpoint, random access addressing seems an invention of computers; we shall discuss this question in section 4.3.2.

(6) The "intelligence" of the central nervous system does not appear meaningfully related either to the size of the brain or to the number of neurons in it. Table 1 shows some data (Blinkov and Glezer 1968). The only really differentiating characteristic so far known is the connectivity. After all, a human brain is smaller and has less neurons than that of an elephant, but has higher processing capabilities.

One may argue whether a similar situation can occur in computers. At present, all computers follow the same philosophy, thus their power is more or less evaluated in terms of their size (memory capacity, number of registers, etc.). By introducing connectivity, different philosophies can be used, and the computer's power might vary independently of their size. It might also be that the use of computers is notably facilitated with a philosophy different from today's approach of instruction-obeying processor + random-access

memory + software system. There is no doubt that the human brain is a quite effective and versatile computer; yet, it seems that it does not use either instructions or addresses. These considerations, expressed here very informally, might be suitable of analytical treatment. The results reported in chapter 6 can constitute a starting material.

T A B L E 1

mean weight of the brain (g)		ratio of brain weight to body weight		number of cells per 0.001 cubic mm of cerebral cortex	
whale	6700	macaque	1/20	mouse	1420
elephant	5200	dolphin	1/38	Guinea pig	525
dolphin	1800	mouse	1/40	rabbit	438
man	1400	man	1/50	cat	308
chimpazee	435	dog	1/250	macaque	215
cat	25	elephant	1/500	man	105
mouse	0.2	whale	1/20000	elephant	69

The evolution of the brain in man has evidently not followed the line of a quantitative increase in size, but the line of an increase in the complexity of the connections between the elements (Blinkov and Glezer 1968). In the last 10,000 years, the size of the brain of man has slightly decreased.

2.5.4 The McCulloch-Pitts correspondence

Certain phenomena in the neural structures present a threshold effect, leading to a relatively easier experimental analysis. Moreover, these effects are typically related to the most distinguishable element of the neural structures, the neuron. Thus, this aspect of the functioning of the neural system was one of the first to be studied, and was schematized in terms of "all-or-none" behavior. A connective with the propositional calculus, and subsequently with the emerging field of computers, was inevitable.

McCulloch and Pitts analyzed this aspect totally. They published (1943) a set of theorems proving that for every logical description of a behavior it is possible to determine a net of logical neurons (axiomatic and simplified models of the biological neurons) that exhibits that behavior; and conversely, the behavior of every net of logical neurons can be described by means of propositional logic. An updated comment on this paper can be found in Fields and Abbott (Ed., 1963) by Arbib.

This correspondence between behavior and logical nets was a milestone in neurology, because it gave for the first time a theoretical tool for simulating the activity of the nervous systems - even that of the highest complexity, the human brain - without the intervention of any vitalistic ingredient. Regardless of the correspondence between the structures of the logical networks and possible structures in the actual nervous system, the behavior of the latter could be, in principle, reproduced in terms of the former. The fact that in practice no single activity of the central neural system has been satisfactorily explained in terms of neural networks (in the sense of modeled within a manageable formal network) is simply the consequence of the structural and functional complexity of the real neural systems. The formal networks are mathematical schematizations of certain threshold phenomena and certain relatively macroscopic structures of neural systems; in the cases in which these aspects play a fundamental or relevant role in the neural behavior, the networks are a useful model; in the cases in which these aspects play a minor role in the overall functioning, the networks do not lead to feasible models.

From the above discussion it may appear that the universal correspondence demonstrated by McCulloch and Pitts between logical expressions and this quite ineffective type of neural modeling is a mere coincidence. Here

the key relevance of the McCulloch-Pitts correspondence to the present work emerges. It is, in another context, the same dualism of functional and structural aspects found so inherent in automata theory. The fact that the structural aspect is thought of as a network of neurons is simply an instance among many others possible. The McCulloch-Pitts correspondence is a mathematical issue. But mathematics is an aspect of psychological mental structures. And psychological structures are one aspect of neurological structures. Thus we can see a deeper universality in the McCulloch-Pitts correspondence, through a series of transformations of which we know very little. Discussions on the relations between biological structures/regulations and logicomathematical cognition may be found in Piaget (1971).

These considerations should make well apparent that the McCulloch-Pitts correspondence is not only relevant to neurology, but also, perhaps even more importantly, to computers. Von Neumann (1948), in talking on a general theory of automata, said on the subject:

The McCulloch-Pitts result ... proves that anything that can be exhaustively and unambiguously described, anything that can be completely and unambiguously put into words, is ipso facto realizable by a suitable finite neural network. Since the converse statement is obvious, we can therefore say that there is no difference between the possibility of describing a real or imagined mode of behavior completely and unambiguously in words, and the possibility of realizing it by a finite formal neural network. The two concepts are co-extensive. A difficulty of principle embodying any mode of behavior in such a network can exist only if we are also unable to describe that behavior completely.

In the light of the work described in this report, I find surprising that these triggering words did not have apparently any resonance in the computer field. But still more impressive observations come after (loc. cit.):

Interpretations of This Result. There is no doubt that any special phase of any conceivable form of behavior can be described "completely and unambiguously" in words. This description may be lengthy, but it is always possible. To deny it would amount to adhering to a form of logical mysticism which is surely far from most of us. It is, however, an important limitation, that this applies only to every element separately, and it is far from clear how it will apply to the entire syndrome of behavior. To be more specific, there is no difficulty in describing how an organism might be able to identify any two rectilinear triangles, which appear on the retina, as belonging to the same category "triangle." There is also no difficulty in adding to this, that numerous other objects, besides regularly drawn rectilinear triangles, will also be classified and identified as triangles—triangles whose sides

are curved, triangles whose sides are not fully drawn, triangles that are indicated merely by a more or less homogeneous shading of their interior, etc. The more completely we attempt to describe everything that may conceivably fall under this heading, the longer the description becomes. We may have a vague and uncomfortable feeling that a complete catalogue along such lines would not only be exceedingly long, but also unavoidably indefinite at its boundaries. Nevertheless, this may be a possible operation.

All of this, however, constitutes only a small fragment of the more general concept of identification of analogous geometrical entities. This, in turn, is only a microscopic piece of the general concept of analogy. Nobody would attempt to describe and define within any practical amount of space the general concept of analogy which dominates our interpretation of vision. There is no basis for saying whether such an enterprise would require thousands or millions or altogether impractical numbers of volumes. Now it is perfectly possible that the simplest and only practical way actually to say what constitutes a visual analogy consists in giving a description of the connections of the visual brain. We are dealing here with parts of logics with which we have practically no past experience. The order of complexity is out of all proportion to anything we have ever known. We have no right to assume that the logical notations and procedures used in the past are suited to this part of the subject. It is not at all certain that in this domain a real object might not constitute the simplest description of itself, that is, any attempt to describe it by the usual literary or formal-logical method may lead to something less manageable and more involved. In fact, some results in modern logic would tend to indicate that phenomena like this have to be expected when we come to really complicated entities. It is, therefore, not at all unlikely that it is futile to look for a precise logical concept, that is, for a precise verbal description, of "visual analogy." It is possible that the connection pattern of the visual brain itself is the simplest logical expression or definition of this principle.

Von Neumann repeated in several instances the expectation that, for complex automata, the description of an automaton is simpler than a literary description of its behavior. Some elaborations on this conjecture, and comments of Burks and Gödel are in von Neumann (1966), pp. 46 - 56.

Here, a simple observation is made. Networks are very effectively used in certain situations such as: the modeling of a neural behavior that can be reduced to the actions of a few neurons; the description of the behavior of a digital circuit composed of a few elements; understanding the interaction of a plurality of units at a high-level description. In these cases, we see

networks to constitute not only a general facility for describing an actual structure, but to constitute also a "language" for describing that behavior. As a matter of fact, often the behavior is expressed in a simpler way by the network than by a corresponding verbal description. Networks continue to be a precise means for describing actual structures also when these structures are very complex; the electric schematic of an entire computer, the telephone network of a city, and a McCulloch-Pitts network for modeling a function of the CNS are examples. But in these cases the networks are not any longer also a suitable "language" for describing a behavior. We can observe very plainly that the ability of a network to be or not a suitable language is determined by the elements of the network to correspond or not to the objects used in our "image" of the system under consideration, when we think of, and understand it. Obviously this fact is related to the processing system of imagery studied by psychologists (see section 2.1.4).

If we consider networks whose elements are not defined in advance, but are made corresponding each time to the objects of our thinking, these networks can constitute a general language for describing behavior. Moreover - a unique and useful feature - they describe also a structure. This approach implies that we might need a subsequent consideration for defining those elements in terms of other elements belonging to a lower hierarchical level.

These networks are not the block diagrams so often used in the most disparate fields, although in certain cases they may appear similar to them; they are not graphical representations of systems already modeled in some other form; they are "machines" that realize the system. Networks as abstract machines are symbolic representations that have the same exactitude and permanency that are peculiar of the formal verbal representations. They are suitable to a variety of syntactic features that are not applicable to the block diagrams originated as graphical representations of already modeled systems.

In summary, what we extract from the work of McCulloch and Pitts is the notion of networks that are simultaneously a description of a behavior and a design of a structure. We consider these networks as a language that describes simultaneously a function and an implementation of it.

Another interesting fact is the following. In applying this approach for modeling processes in the form of "abstract machines", we find (see

chapter 6) that the description of these machines, especially for complex processes, is typically simpler than the description of the processes the machines implicitly represent. It is a suggestive correspondence with von Neumann's expectation, above mentioned, on the complexity of automata and their activity.

We conclude this section on the McCulloch-Pitts correspondence with the last words recorded on the work guided by McCulloch in this direction, p. 339 of an M.I.T. (1968) Quarterly Progress Report:

Just as a universal Turing Machine can be made specific for the computation of a particular number by having a portion of its tape serve as a program, so can a "universal net" of N neurons be made specific to embody any net of N neurons (with or without loops), by a proper encoding of its inputs. . . .

Neither the logic of relations, nor the theory of neural nets is fully developed. We expect both to bear fruit in due season, and have only reported their present flowering.

At the time in which the research reported in the above quotation abruptly stopped, not far from that place, but unfortunately without communication, in a different context, the work reported in the present report was starting.

2.6 OUTLINE OF THE APPROACH TAKEN

2.6.1 Synopsis, part 1

We started in section 1.1 with the realization that limitations in the use of computers do not come from technological difficulties but from the cumbersomeness of communicating with computers, and depicted the situation graphically in figures 1 and 2. In section 1.4, the need for a global approach was expressed, which calls for an examination of the psychology of the user and for a search for possible processing implementations. After a tour of the findings of psychologists, automata theorists, and neurologists, we come back to the consideration of a global approach. Its development is outlined in the following.

In the first place we observe that the approach used today, as depicted in figure 2, seems affected by a remarkable sequence of successive transformations. A human being develops a process, to be later given a computer, in terms of his mental structures; these structures have a variety of forms,

2.6.1

and are open to continuous re-arrangement. Fundamental in the development of a process is the geometrical intuition. Also in his most abstract activities, a human being is bound to use the structures that have been formed at the sensorimotor stage; all the gestures and sketches we are familiar with, and the imperceptible muscular movements found by psychoneurologists prove this abundantly. The most general characterization of thinking, in the context of using computers, seems to be a mechanization of abstract objects. Giving the term a very broad sense, we can view these mechanizations as abstract machines.

Then, the abstract machine in the user's mind is transformed into an equivalent verbal structure -- the process expressed in a programming language. Although in some cases the programming language allows an almost isomorphic transformation of what the user has originally in his mind, in most cases a complete remodeling of the process is required on the part of the user before the transformation can be accomplished. This is a user effort. The fact that this effort can be decreased by a long training is not a very desirable solution. Since a professional is necessary at present, this training is precisely one of the inconveniences, and it is questioned here. Possible effects of this training in human behavior will be commented on in section 5.1.2.

Subsequently, this verbal-structured representation of the process is transformed further, through other intermediate languages, until finally it matches the characteristics of a given, real machine -- the computer hardware. We have already noted the further effort encountered by the user for interpreting the work of this machine during the debugging phase.

Then, we observe that, in the outlined interpretation of today's computers, we start from a machine (abstract, in the user's mind) and we end with another machine (physical). Now, the question comes spontaneously whether a more direct connection between the two machines could be possible, without going through all the described transformations.

The path used by today's computers does not seem to have a universal necessity, as it can be inferred from the different approaches seen in the different contexts examined in the previous sections. It is simply a happening. Men are accustomed to give verbal orders to subalterns; first digital computers were elementary and made by simple on-off devices for which simple verbal instructions were very appropriate. On the other hand, we can

see, for instance, that analog computers were programed in a different way.

For the possibility of a more direct connection between user and computer, we see the necessity for a first change. The real machine mentioned in the above description has a highly fixed structure (or, in other words, little connectivity); it is precisely the verbal structure which provides the matching between the variety of the user's abstract machines and the rigidity of the real machine. If we intend to eliminate, or reduce, this matching process, we need the real machine to be capable of assuming the variety of structures of the mental machines. We have already seen cellular spaces (section 2.4) capable of becoming any conceivable automaton; we saw the human cortex (section 2.5.2) thought of as an organ capable of becoming a variety of other organs. Here we need a substratum capable of becoming the various abstract machines of conscious thought in the context of the processes we give a computer.

In fact we need a substratum with dual aspects or, in other words, two isomorphic substrata in two different domains: one in a symbolic domain, to play the role of a language in which the user can externally express his mental machines; and another in a physical domain which, when molded by a production of the first substratum, implements an actual, operating machine corresponding to the original user's mental machine.

In regard to the nature of this dual substratum, it can not be precisely as the cellular spaces discussed in section 2.4, because they do not match the user's natural way of thinking of processes, and they lead to inefficient implementation. A similar situation can be expected in regard to the human cortex (a neurological structure, as opposed to the psychological performance), although we know too little about its actual working. However, in the general notions of automata we see the possibility of defining such a substratum. Automata seem to extract what is essential in the processes, and they have a double aspect, one functional (which can be used for the language role), and one structural (which can be used as a design for actual implementation). The levels at which the automata are conceived will be several, in a hierarchical relation, as our thought is; in section 2.5.4 the subject was preliminarily discussed.

Along these lines we will arrive at an actual definition and implementation of the desired dual substratum. In the next section, the structure/function duality will be discussed in some detail, and in section 2.6.3 the

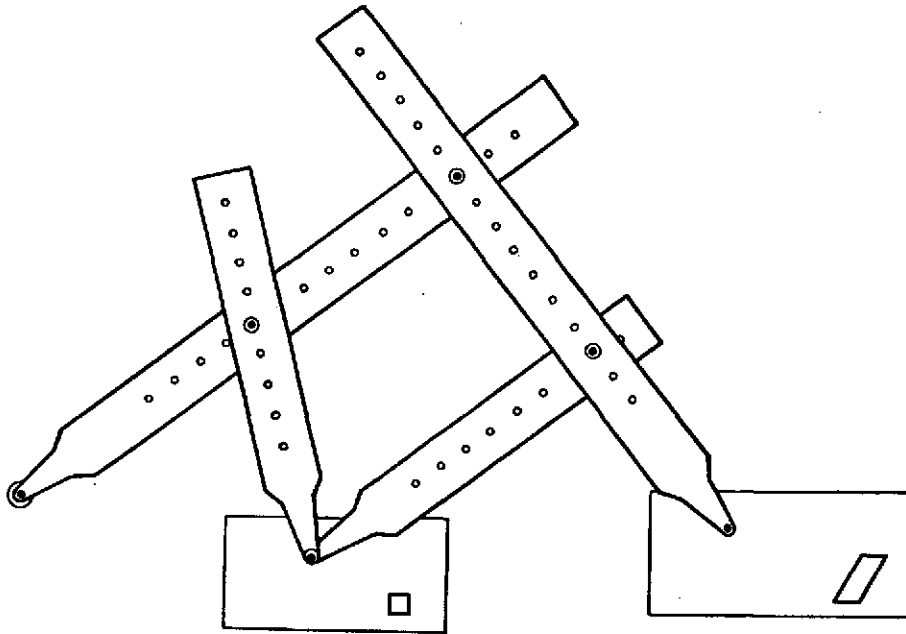


Fig. 6 - Transformation of drawings by means of a pantograph

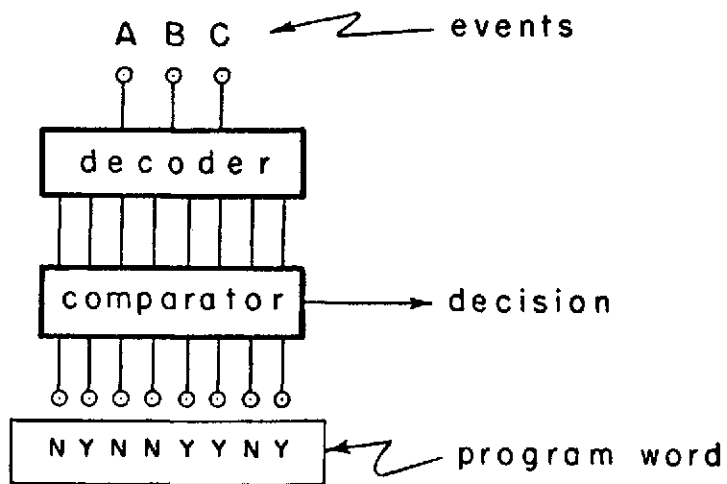


Fig. 7 - A programmable decision device

synoptic outline will continue from other viewpoints.

2.6.2 Function and structure

When a process is modeled in the form of a sequential machine, we obtain a rigorous definition of the process, but also we find in our hand an abstract structure that performs that process. When a neural behavior is modeled in the form of a McCulloch-Pitts network, an explanation of that behavior is found, but we find in our hands also a hypothetical machine that produces that behavior. When a certain performance is obtained by means of a cellular automaton, we have in front of us both a functioning and a structure that produces it.

From all these examples, we see that one way for our thinking to define a process is that of devising an abstract machine that mechanizes it. It is evidently a special-purpose machine, in fact, even more, it is a completely tailored machine; we prefer to say that it is the process itself. It is also the most effective machine at our disposal for that process; if we could think more effectively, we would accordingly sketch that machine. It is clear that our thinking has indissolubly embedded a functional and a structural aspect; that the different forms in which we can model our cognitions can differently enhance one or the other aspects; and also, that for complex processes we tend to enhance the structural aspect in order to use that powerful capability called geometrical intuition. In accordance with these considerations we will form our substratum.

The interesting point is that all such abstract machines (if we avoid distractions and illusions) are physically realizable, given appropriate means. This is simply the consequence of our thinking to be developed with years of adaptation to a world that we call physical and say follows certain laws. The cleverness and efficiency of these machines depends on the familiarity of the user with the process in question, on his skill, and on the teaching he can have from outside. We are looking for a magic substratum that can make these imaginary machines real; let us start with two preparatory examples.

Let us suppose that we have the task of reducing the coordinate scales of a collection of drawings. In this case somebody has taught us the mechanism of the pantograph. We take four bars (a standard element in a mechanical context), assemble them as in Fig. 6, and proceed in the well known manner. Note that we can produce the process even without knowing

rules of geometry and without the need to express any word. If we had a universal machine in which we could command all movements, a considerable effort would have to be spent for finding the necessary geometric rules, for applying them to the different instances in the drawings, and for communicating all that to the universal machine. The example, because of its mechanical nature, might not appear relevant to the present context. I think, instead, that it is very helpful for pointing out certain general points. In particular, (1) it is a simple and revealing example of the effectiveness of a special-purpose structure in producing a complex process (this strategy seems completely neglected in today's computers); (2) it is a suggestive example of how data can be properly "channeled"; the four bars constitute not only the operational structure, but also the addressing device (this is a psychological preparation toward the liberation from the "random access" trap).

As a second example, let us suppose that we have to handle a variety of complex decisions. The approach used in today's computers is to model them in terms of elementary tests. But a global approach can be very well taken by modeling them as a single logical function, and implementing it with a corresponding device. Fig. 7 shows a symbolic frame for thinking of decisions, composed of a decoder and a comparator (standard elements in a computer context). There are three types A, B, and C of events that may or may not occur (YES or NOT, respectively). A decision has to be taken depending on particular compositions of those occurrences (Y) or not occurrences (N). A word composed of Y and N, in accordance with a certain morphology, is presented to the comparator, and the decision follows in accordance. Note that the morphology of this word is suitable of meaning; for instance, the first N of the word means that the decision should not be made when there is no occurrence of any A or B or C; the last Y of the word means that decision is made when all A, B, and C occur; a Y in all even positions of the word means that A should always occur; an N in the first four positions means that C never should occur, etc.

The structures of these examples can be thought of as equivalent to parameterized programs; each is capable of a class of different processes. The pantograph is capable of different reductions and isomorphic deformations of drawings, by inserting the shafts in different holes of the bars. The decision device performs different discriminations, by changing the word inserted into it.

In the two examples above, the procedure was elegant because we started from a mechanism already invented. But we cannot expect that this will always be the case; nor are we interested in an approach in which the user has to make each time a new invention. Fortunately, it is not so. The only assumption taken is that the user has the process in question clear in his mind (this is not really a limitation; it is a common experience that it is preferable not to start action if one is not clear what he wants). This, obviously, does not exclude the fact that for complex processes clarification develops gradually through trial and error. We proceed as follows.

If a user understands a process, inevitably he has a mental image of it; thus he will be able to sketch this visualization. This sketch, regardless if graphical, mental, or verbal, is an abstract machine; at this point, it is the process in its entirety. This abstract machine will be composed of parts; to them the user applies again his visualization and geometric intuition powers, with new abstract machines resulting. This visualization if repeated recursively, until the parts of the machines are elementary in the given context, or have known implementation. Insofar as we remain in the domain of abstract machines, always it will be possible to express these abstract machines in some form, for instance with sketches and words.

The McCulloch-Pitts correspondence shows that when these abstract machines have a certain formality, they can be implemented by realizable devices. Von Neumann even argues that, for complex processes, such machines can be simpler than a description of what they perform. The key for the practicality of this approach is to stop the succession of hierarchical visualizations at the point where the user stops to be interested for his present purposes; in other words, the dual substratum we are seeking should be at the level of the user's interest.

In sections 2.3 and 2.4 we saw several substrata for implementing automata; their common characteristic is to be at a very elementary level. The motivation for this is in their objectives: analysis, classification, search for universal bases. For these tasks, mathematical simplicity and homogeneity are essential; efficiency is irrelevant. Here, the objective is to represent mental images; this was the reason for the previous review of psychological theories of thought.

Conventional general-purpose computers are also substrata for processes; but they are unsuitable for being molded by our mental images. The indirect

2.6.2

way referred to in section 1.2 leads to the inconveniences discussed in section 1.1. The rigidity of present computers has been deplored repeatedly, especially in the past, before the present dominance of software. For instance, Bauer (1960) says "almost all the needs can be summed up in one short comment: there is a need for computers which can adapt to problems. Up to this time computer customers have found it necessary to adapt their problems to the computers". Estrin (1960) went further than a simple complaint; he proposed a symbiosis of a fixed conventional computer and a variable structure consisting of an inventory of substructures selected in accordance with the frequency of their use. But it is neither simple nor efficient to have in advance an inventory of all the structures that may be needed for all problems; a user might wish a new piece that never was considered before. We can see also that the approach of analog computers is not suitable for our objective; in analog computers, the processes have a fortiori to be modeled in terms of the available devices (e.g. integrators, scalars, etc.); here, we need devices as suggested by mental images.

In this study, in regard to the level of the substratum, we take the same position as that of psychologists in their interpretation of human thought: new structures are formed by grouping and integration of previously developed structures, starting from a genetic level of the substratum. This substratum should permit constructs equivalent to verbal structures, processing of words in accordance to a syntax; and constructs equivalent to images, complex objects that are inclusive of data and functions, treated as a whole, of which parallel and special-purpose structures should be among the possible applications. Fundamental in this substratum should be the spatiotemporal frame, in order to take full advantage of our geometric intuition.

2.6.3 Synopsis, part 2

If a symbolic substratum is available in which the objects of our thought can be molded, and with which objects our mental images of the processes can be described as abstract machines; and if a physical substratum, isomorphic to the first, is also available that can implement those abstract machines, we can look again at Fig. 1 with a new interpretation.

Point A represents the mental abstract machines in the form of which the user can think of the processes. Point B is the representation of those machines in the symbolic substratum. And point C is the actual information that

should be given the physical substratum in order for it to be molded in the form of those abstract machines. The three points are distinct because they belong to different domains, the mind, a representation, a hardware; they are expressed in different media; but they are structured in the same way. Thus, we can expect that the transformation of one point into the other will not constitute a great effort. This approach corresponds to the intuitive idea (geometrical intuition!) expressed at the end of section 1.1 to shrink Fig. 1 into a single point.

In practical terms, rather than making first an effort to frame the processes into stereotyped sentences, and then require the computer to make another effort to re-transform those sentences into a machine behavior, we focalize our attention to the natural way of seeing a process as an abstract machine, and then we ask the computer to copy it. From all the previous discussions, we can be certain that the language of abstract machines is broader, and more widely suitable to different applications, than is a single phrase-structure language. A proper choice of control devices and machinery has made it possible for the majority of individuals to run tens of times their walking speed and to extricate themselves in remarkable complexities such as the automobile traffic in Los Angeles and Rome, by using sensorimotor structures developed since early stages. It seems to me that it should be also possible to choose proper control devices and machinery to make it possible for the majority of individuals to obtain by themselves, with the help of computers, symbolic processes orders of magnitude more complex than those mentally affordable, by using imagery structures developed during life.

In accordance with all the discussions in sections 2.1 and 2.2, the actual materialization of point B will take advantage of visual forms of representation. We can hope that the clarity peculiar to state diagrams and the properties analyzed in graph theory can be joined in the development of an effective graphical language. These graphical structures will be embedded with verbal expressions for their unique power of characterization and their complementary features.

Having set forth the lines of our study, we close the preparatory part (the first two chapters) with an exercise of a graphical representation of this report. From introspection and from psychologist's analyses we know of the existence of several facets in thinking. We know also that processes can

be implemented in different ways, e.g. Turing machines, analog, and digital computers. This situation is visualized in the form of blocks in Fig. 8. There, a dotted line indicates the course of the discussion. We started by recognizing certain inconveniences in communicating with present computers (chapter 1); then we went wondering what the user has in mind (chapter 2). In chapter 3, a substratum for abstract machines is formulated, in chapter 4 an isomorphic physical substratum is outlined, the programming language is discussed in chapter 5, and finally, in chapter 6, results are presented.

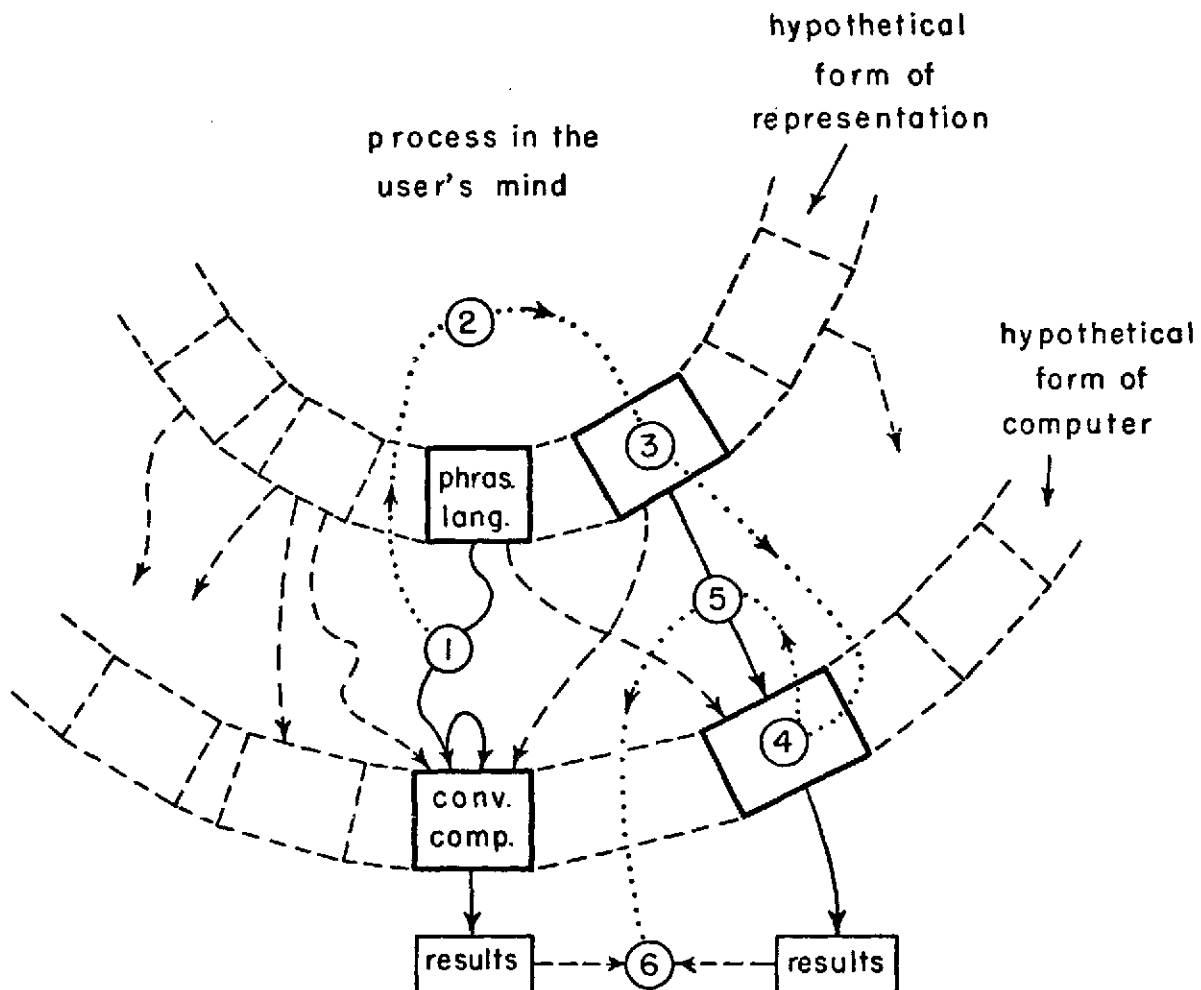


Fig. 8 - Graphical visualization of different approaches to computers

Chapter 3

The Formulation of a Symbolic Substratum

In the previous chapter we have developed the background for a symbolic substratum, in a spatiotemporal frame, in which abstract machines can be developed, corresponding to the user's mental images of the processes. A substratum that can also permit an isomorphic physical substratum, such that the abstract machines delineated by the user in the symbolic substratum constitute also the design of an actual implementation of the machines in the physical substratum.

Below, some preliminary characteristics are discussed; the formulation of the substratum is carried out in sections 3.2 and 3.3; and in the following section the substratum is compared with the other systems that are used for describing processes.

3.1 PRELIMINARIES

Sequentiality - Mental processes appear basically sequential. We analyze information sequentially; we accomplish tasks sequentially, often in a sort of time sharing, which itself is a form of sequentiality. Sequential physical implementations, as is well known, can accomplish large tasks with small means. There are enough justifications for starting with a sequential structuring of the abstract machines.

We note also that thinking acquires a special power by means of its capability of keeping extra items at the conscious level, in a kind of background, during the sequential performance of mental processes. We will give our abstract machines an equivalent capability.

Time - Sequentiality implies a time frame. Certain precautions are necessary in order to avoid ambiguity and instability, in regard to both logical consistency and physical delays in implementation. In automata theory two

solutions are usually taken: the synchronous approach, and the asynchronous one with unit-delay elements properly introduced. Here we take the following approach that shares the interesting features of both preceding ones.

Time is quantized in intervals labeled $\dots i-1, i, i+1 \dots$ (Fig.9). The length of these intervals, in abstract or physical sense, is irrelevant. At the conjunction of two adjacent time intervals there is a change region. The characteristics and values relevant to our machines are defined for each interval i , and we consider them as remaining constant within each interval. Changes can occur in the change regions, but we organize things in such a way that we do not need to be involved in how they occur in these regions.

This approach will be used both at the macroscopic level of the process modeling, and at the microscopic level of the actual execution of the specific data transformations.

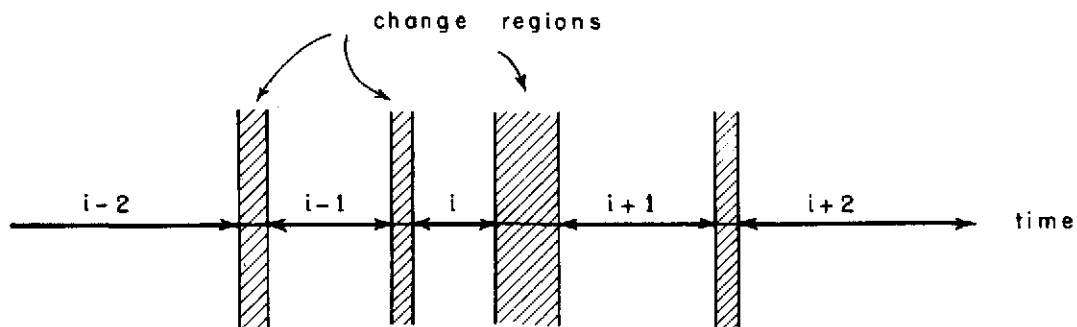


Fig. 9 - Quantization of time.

States - In observing our sequential way of thinking, we can say that in fact we switch from one image to another, from one consideration to a subsequent one, from one viewpoint to another. There are, of course, interesting variations from individual to individual: some data are given in section 5.1.3. It is a common fact that under an external stimulus, such as receiving an unexpected word or hearing a telephone ring, we are able to change suddenly the entire subject of our thinking. There is no need of further examples for considering the notion of state of mind, or simply the "state", to be an element appropriate for a psychological substratum. The notion of state is

probably the most important building element of automata theory. In several different contexts, the notion of state is also applied to physical implementation of discrete systems, such as computers. As a conclusion we assume the notion of state as a building block of our dual substratum.

The notion of state can be applied at many different levels and in reference to many different objects of the discourse, with a consequent very different usefulness, that a clarification of the subject is appropriate at this point. As a specific example, let us consider three possible applications of the notion of state for a given system - a digital computer.

In the first application, we consider all the independent, steady-state logical levels that are present in the entire structure of the computer. As is well known, we end up with an astronomical number of possible states. States defined at this level might be of interest for analyzing the local functioning of a particular circuit of the computer, but they are useless for understanding the functioning of the computer and for analyzing the processes executed by that computer.

In the second application, we consider the computer as composed of parts treated as units; each of these units can be in a finite number of functional states, and we consider as the state of the computer the cartesian product of the states of its parts. In this case, the number of states might be manageable, and the consideration of these states might also be of interest for analyzing the interactions among the parts of the computers. However, these states are still useless in the analysis of any problem under execution, because they refer to machine conditions and not to problem characteristics.

In the third application, we consider the entire computer as a unit, and assign to it three states: the quiescent state before a problem is received, the execution state, and the quiescent state after execution with the output available. Here the number of states is too small to give any insight into either the functioning of the computer or the characteristics of the problems executed. The only interest for such a state definition would be for controlling an outside device that gives the problems to the computer, and that acquires the output results.

From these examples, we see that the number of states varies tremendously according to the level at which the states are defined; but we see also that it is not simply their number that affects their usefulness; in no one of the

above applications, states are useful for our goal of modeling processes. The notion of state becomes extremely useful for modeling, if the states correspond to our visualizations of the processes; in particular, if they correspond to our sequential consideration of the different phases of a process. In this case, the number of states does not affect notably the practicality of the model, provided that all the states together form a structure of the process as it is understood by the user, and that the majority of these states have a role meaningful to the user.

Transition - If the notion of state is used, we will need also the notion of transition from one state to another. This means that in our machines the prescription of how to transfer from the present state to another future state has to be a fundamental machine element. These prescriptions are called transition functions in automata theory. They exist in present computers only in the rudimentary form of jumps.

Because our substratum should have both machine and psychological orientations, the transition functions will be derived both from automata theory and from observations of people thinking (see section 5.1.3).

Data transformation - If a process is considered, it is because of the need of transforming some information. Therefore, at least in some "state" of our machines, some data transformation should occur. We will keep the term "data transformation" in order to treat comprehensively the different activities that might occur, such as those related to analytical functions, logical functions, data reduction, coding, selecting, formating, etc., all viewed in a global Gestalt approach.

Input-output - Sometimes we think in complete isolation (off line, in computer terms), and sometimes we think in interaction with the environment (on line, using the same analogy). It is a common occurrence of life that during the process of our activities we receive information from outside, and we have to give information to the outside. Since our abstract machines are intended to be a natural outgrowth of the user's mental images, we will provide these machines with a general capability of acquiring data from, and producing data to the outside at any moment of their process, without being bothered by any constraint or delay.

All the above preliminary characteristics are chosen a priori for our substratum as appropriate for both the psychological and the realizable-machine aspects. It is difficult to say whether the considerations of this section constituted the initial requirements for the formulation of our abstract machines, or the a posteriori justification after the formulation, or a cyclical adjustment and improvement of the substratum. Probably, some of all of the above is true.

3.2 - ELEMENTARY MACHINES (FSM AUTOMATON)

3.2.1 - Symbolic formulation

The primitive elements considered are:

- (i) a finite set of process variables x_r , a generic subset of which is indicated as X_q ;
- (ii) a finite set of input data u_r , a subset of which is indicated as U_q ;
- (iii) a finite set of output devices, or control storages, z_r ;
- (iv) a finite set of labeled process-states s , where a state is defined by:
- (v) a function F_j which produces new values for a subset X_a as a function of the values in subsets X_b and U_c ,
- (vi) a function T_j which produces a label s (the next state) as a function of the values in subsets X_d and U_e , and
- (vii) a prescription R_j for routing some variables x_r to some of the output devices, or control storages, z_r .

An abstract machine is defined as a finite set of quadruplets

$$\left[I_j, F_j, T_j, R_j \right]_s \quad (3.1)$$

$$s = 1, 2, 3, \dots k$$

where I_j is the prescription of an input subset $U_j = U_c \cup U_e$,

F_j , T_j , and R_j are as defined in (v), (vi), and (vii), respectively, and s ranges through the k states of the machine.

The variables x_r and their subsets X_a , X_b , and X_d are implicitly defined by the functions F_j and T_j . Without loss of generality, we suppose in the following that x_r and u_r belong to a finite alphabet of integers from 0 to 2^m .

Such a machine, as far as its internal behavior is of concern, can be functionally represented by the two expressions

$$\begin{aligned}
 X(i+1) &= F_{s(i)} \left[X(i), U(i) \right] \\
 s(i+1) &= T_{s(i)} \left[X(i+1), U(i) \right]
 \end{aligned}
 \tag{3.2}$$

where the symbols have the meaning indicated before, and i is the time as formulated in section 3.1. For each value i of time, $s(i)$ and $s(i+1)$ are the present and next states, respectively. It is probably appropriate to recall that, here, the states are "process-states" and not states of the variables; a state refers to a quadruplet (3.1); in successive times i , the state (a quadruplet) may remain the same, but the variables typically change. Further discussion will be found in section 3.4.1(6).

Such a machine can be represented also in the form of a state diagram, by means of proper conventions. In chapter 5 a set of conventions are given, and in chapter 6 many examples are shown. A machine so formulated is referred to as a Finite State Machine (FSM); capital initials are used to distinguish it from the otherwise formulated finite-state machines.

3.2.2 - Structural formulation

Let us call logical network, or simply network, any logical network that, regardless of the devices used and the level of the functions considered, corresponds to a logical activity in the sense discussed in section 2.5.4. If F_j stands for a logical description of an activity j , and N_j for a logical network performing that activity, we can thus consider the mapping

$$F_j \longrightarrow N_j \tag{3.3}$$

Let us now consider a very large (finite) logical network composed of operating and storage elements. For simplicity, and without loss of generality, we assume binary quantities. Such a network can be regarded as a giant, unmanageable, finite-state machine with a very large number of states and input signals. The state diagram of such a machine has such a complexity that it can be considered undescribable; we call it a total state diagram. Let us divide the input signals into two categories, which we call v inputs and p inputs. For each set of values at the p inputs, a particular "compon-

ent" of the total state diagram is selected, while all the rest disappears. Such a component can be regarded as a specific computation on variables connected to the v inputs.

If there are m of such p inputs, we can look at the ensemble of the p inputs as a word W of m binary digits. All the possible patterns for these digits form a set of 2^m words. Let us call L the subset of these words that correspond to meaningful logical networks. We can thus consider, in the subset L , the mapping

$$N_j \longrightarrow W_j \quad (3.4)$$

where N_j is a logical network performing an activity j , and W_j is a digital word implementing that network. In the sequel, we will call a programmable network (PN) a logical network as described, where words W_j can implement specific networks N_j that perform activities F_j . We consider the subset L as the language of that programmable network, and the mapping

$$F_j \longrightarrow W_j \quad (3.5)$$

as the semantics of that language.

If the objects and the structures of the network are made correspond to those that are used in the descriptions F , an encoding can be chosen for the network such that the descriptions F_j can be mapped into the words W_j by direct transliteration.

A programmable network PN, as previously defined, is embedded with a set of internal variables x_r (Fig.10). By the term embedded it is meant that the variables are expressed in some digital form and are stored in some elements of the network where they can be read and written by some inputs v and outputs of the network. The p inputs of PN read digital words W_F and W_T in a storage P . Some of the v inputs and some of the outputs of PN are connected to input lines ξ and output lines ζ , respectively, through switching devices that respond to digital words W_I and W_R , respectively, in P .

We give such a structure a control for selecting quadruplets in P. During each time interval i , words W_I and W_F of the selected quadruplet are connected to the input switches and to the p inputs of PN, respectively; at the first change period, the network is activated, F is performed, then W_T is substituted at the p inputs, a selection of a quadruplet is determined for the future interval $(i+1)$, and finally W_R is connected to the output switches. In the next time interval, the newly selected quadruplet will perform the same way.

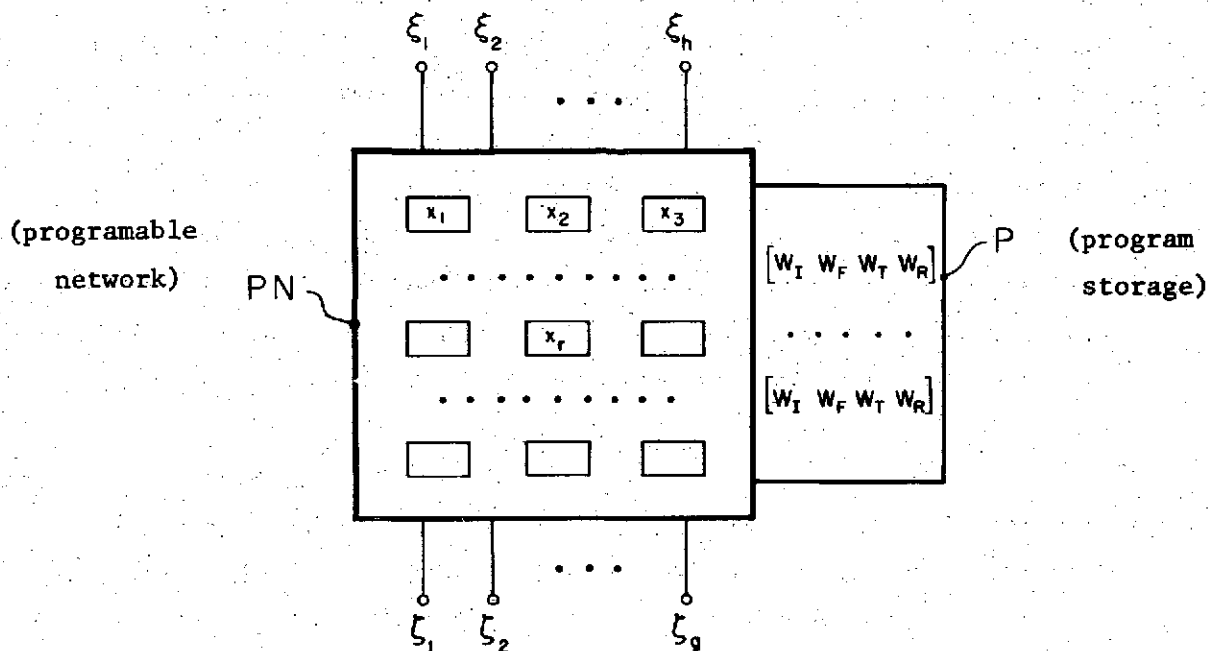


Fig. 10 - The F S M automaton

In the realm of the language L of PN, this structure parallels the symbolic automaton defined in section 3.2.1, in the realm of the language there tacitly implied. The symbols I , F , T , and R there considered correspond to the symbols W_I , W_F , W_T , and W_R used here. In both cases, when specific and proper values are given to these symbols, specific machines are defined. With the first formulation, the machine is in a verbal-like structure; with the second formulation, it is in an imagery-like structure.

In accordance to this correspondence, the automaton now formulated will be called also a Finite State Machine (FSM).

3.3 COMPOUNDED MACHINES (CPL AUTOMATON)

The elementary machines just defined (the FSM automata) constitute a sufficient, or appropriate, modeling frame only for simple processes. A more powerful frame is necessary in cases such as:

- (1) when several processes have to be dealt with simultaneously;
- (2) when our mental limitation requires to attack a complex process in smaller parts separately;
- (3) when the size of the process exceeds the assumed size of the FSM automaton.

In these cases, a compounded machine is developed, by grouping a plurality of elementary machines.

3.3.1 Symbolic formulation

Each component FSM is defined by a set of quadruplets (3.1), and represented by expressions (3.2). We will give a name to each FSM in order to make communication between them possible. Corresponding extension should be given to the elements I, F, T, and R.

Obviously, to make compounded machines treatable and consistent, a new grammar has to be formulated, and possible limitations spelled out. But we will not go further in attempting to define the rules and the constraints of such a grammar in a symbolic context.

3.3.2 Structural formulation

Let us connect an FSM automaton, as defined in section 3.2.2, with a memory in the following way (Fig. 11). During intervals i , the content of PN is removed by a control, packed in the form of a page of data, and stored into the storage medium of the memory, through its input lines.

A page here is a virtual replica of an FSM automaton. It includes the present set of x_r , the state label $s(i+1)$, and the set of quadruplets describing the FSM; these quadruplets may be substituted, in the page, by the name of the FSM, if they are stored elsewhere (as will usually be the case in the isomorphic physical substratum). Although the page is thought of as a compressed package, it is a "structured" set of data, in the sense that the detailed information of the allocation of each variable in PN is preserved in some form.

As soon as the FSM automaton has been emptied, the control transfers into it a new page from the memory, through the memory output lines, and allocates data into PN according to the above-mentioned information inherent to the structure of the page. In this way, a large number of FSMs can per-

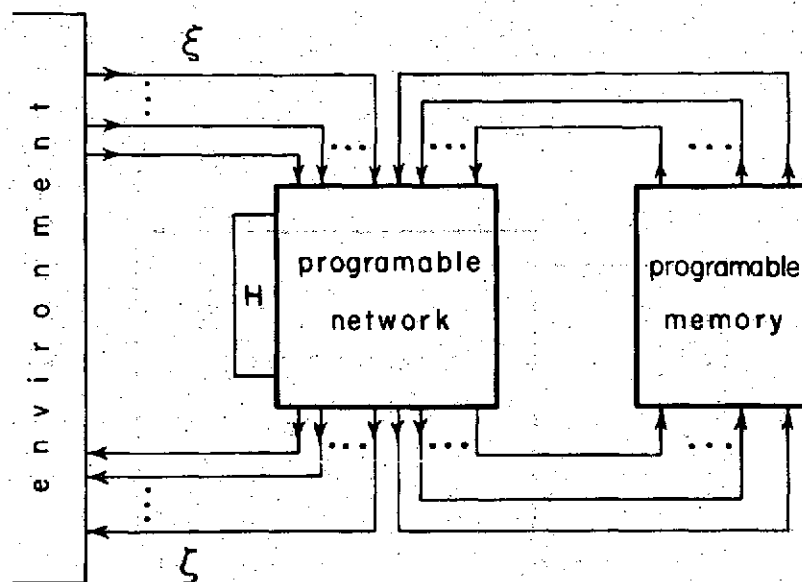


Fig. 11 - The CPL automaton

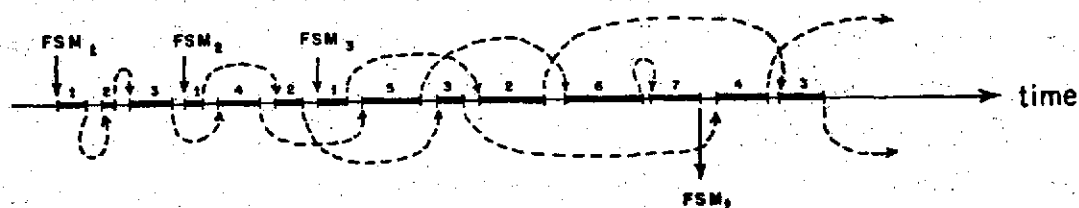


Fig. 12 - Cycles in PN

form their processes, through the circulation of the pages between PN and the memory, in a time-sharing fashion. All the change periods of the several FSMs are made contiguous in the PN, as diagrammatically indicated in Fig. 12, and are called the cycles of PN.

The storage medium of the memory is structured by the injection of pages, and by the content of control storages z_r (iii in section 3.2.1). The pages injected by the FSM automaton form a sequence of blocks, typically of different lengths. This sequence is segmented and structured in accordance with the content of certain control storages z_r . The path of the pages between the input and output of the memory is a consequence of this structuring. The content of the control storages, and thus the structure of the memory, may change at any cycle as a consequence of routing R by part of some FSM.

The continuous, automatic circulation of the pages gives not only the possibility for many FSMs to use the same PN in time-sharing, but, not less important, allows a continuous dynamic reshaping of the memory substratum. Each page can independently change in size during processing in the successive circulations. PN can generate new pages that become inserted into the memory segments. Pages can eliminate themselves simply by not circulating.

The page is a data structure that automatically modifies itself during processing as a consequence of the Is (introduction of new data), the Rs (routing of data to outside or displacement of data), and the Fs (data transformations). These pages, in their actual (not virtual) form, are considered by the user in his dealing with the process, and are manipulated by the programmable network in the execution of the process. It is this identity of data structure, as seen by the user and by the computer, that contributes to make simple the man machine interaction.

The array of storages x_r in PN has a replica, indicated as H in Fig. 11, also embedded with the programmable network. Storage H can copy a page currently in PN, can transfer a stored page into PN, and can take part in the operations performed in PN. The content of H is not removed during the circulation of the pages. In this way, the several FSMs in process can exchange data and interact through H. Storage H has a role equivalent to the

psychological background noted in section 3.1 in regard to sequentiality in thinking.

The input-output connections with the environment allow each page to acquire new input data u_r and routing data to outside output storages, at each circulation. From the viewpoint of each FSM, the entire structure is as if it were dedicated solely to its own process; thus each FSM is treated as indicated in section 3.2.

An FSM (a set of quadruplets) can be implemented by several pages, which simply carry the name of the FSM and the present state; a page can go through several FSMs by changing the FSM name. Variables x_r may refer to names of FSMs and pages. A specific implementation of the language for describing these structures is given in chapter 5. The dynamic behavior of these structures can be best represented, again, by means of state diagrams, by using proper conventions.

A compounded machine is developed by defining several interacting FSMs and related sets of pages. A complex process is modeled as an interplay of processing structures (the FSMs) and data structures (the pages), developing them as suggested by the mental images that the user has of the process in question.

A variety of coordinates are available along which to model a process: a parallel array of data and operational structures in PN; a parallel array of pages in the memory; a sequential pattern of a page during its circulation; a sequential evolution of arrays of pages in the memory; and a sequential interaction with systems in the environment.

It can be noted that the compounded machines have been introduced in a structural form; the corresponding symbolic form is not presented as it was for the elementary machines. This is because it is only through geometrical intuition that we can manage complex behaviors. If we had formulated a compounded machine in a purely phrase-structure form, much more effort would have been needed, and at each step we would have had to take recourse to local spatiotemporal images for understanding what is going on. For this reason we prefer to start from an overall image in structural form and, when appropriate, to take recourse locally of verbal forms such as the words of the quadruplets and names. This corresponds precisely to the alternation of images and words that occurs in thinking (section 2.1.4).

Because of the peculiar features of such a machine — the organization of data in structured pages, their automatic circulation, and the loose nature of both the PN and the memory — it has been referred to as the Circulating Page Loose (CPL) automaton.

3.3.3 Discussion

What has been described constitutes a symbolic substratum that implements the characteristics outlined in section 2.6. There are three regions (Fig.11), the programable network PN, the programable memory, and the environment; each has different orientation, and is suitable of different structuring. PN is a region where the user can develop structures of data and of operations; it can be considered as corresponding to the psychological short-term memory and local images. Through this region, patterns of symbols (the pages) circulate from a dynamically structured memory (the page memory) that, in fact, implements a large number of virtual PN regions. In actuality, the multiplication of PN regions occurs sequentially, but there is a complete interaction between PN and the memory, that both can be considered as forming a larger, complex machine(s). This ensemble corresponds to a kind of multiple visualization and to an intermediate memory employed by the user in his development of the processes. Each time a page is in PN, it can exchange data with the environment, which includes the systems with which to interact, and with other regions of the memory.

The genetic level of the substratum can be the single digit, since both PN and the page memory are programable. The working level of the substratum is that of the primitives listed in section 3.2.1. Operational and data structures are treated as a whole, by means of the words F and T, and the names of pages and FSMs. In PN, operational and data structures are developed conjointly, as has already been seen in many forms of automata, and these structures correspond to the images that the user forms of the operations; situations such as that of Fig. 3c are implemented, rather than sequences of steps such as in Fig. 3a. In sum, it is a standard substratum in which the user can develop the structures that he feels are appropriate for the problem at hand. What we call here an abstract machine is, in essence, a spatiotemporal dynamic representation of a process, in terms of objects constructed by the user in this substratum.

Note that the language for describing these machines is based on a syntax very familiar to all users: the rules of a pseudo-spatiotemporal frame, against which the programmer can check the consistency of what he is developing. In fact, this frame helps the conception of the programs and guides their construction. This is particularly apparent when compared to the instruction or statement listing of conventional programs.

Another help to the user comes from the hierarchical structure of these abstract machines. First, the overall structure, the strategy for the model of the process, is conceived in terms of FSMs and pages. Then the state diagrams of the FSMs are developed. Finally, the operational structures, and the inflected words that make up these structures are constructed in detail (see chapter 5).

The basic point of this approach is to invent a machine (in the broadest sense) appropriate for each case, process, task. In other words, one has not to imply a set of given devices, on the basis of which appropriate algorithms or procedures should be prepared, but has to prepare a specific device that will do the desired activity as its natural response, and, if it is the case, following specific methods requested by the particular application.

The devices to be invented are "abstract materializations" (the contradiction is purposely made to focalize the different aspects of the point) of what the user has in mind — a consistent, realizable description of how he thinks the process can be produced in an abstract world. Now we can foresee from a general viewpoint the difference that results in programming. Conventional programming consists of transforming activities in order to become procedures executable by given devices. Here programming consists of synthesizing devices for which the given activity, precisely in the form preferred by the user, is an executable procedure.

Given the developmental character of these machines, it will be possible to use implicit description of them. A program may consist of a very concise description of a generating machine; then, this machine will construct the entire system of FSMs and pages necessary for the execution of the intended process. Here we come close to self-organization; but actually to perform self-organization it is necessary to define tasks, criteria of evaluation, and interacting environments, all topics that are not included in the present report.

3.4 COMPARISON WITH OTHER FORMAL SYSTEMS

3.4.1 - Comparison with the commonly formulated finite-state machines

The "FSM" is based on the same notions of state, input and output symbols, and transition function as are the "finite-state machines" which can be said took first organic form with Moore (1956), and which are now well established in automata theory. In recognition of this fact, the automaton defined in section 3.2 has been indicated with the same terms. However, there are differences; to help avoiding possible confusion, we use capital initials, Finite State Machine (FSM), for our version.

The differences between the FSM and the finite-state machines of the literature are of two types: in application and in formulation. Finite-state machines are typically used for modeling the behavior of the simplest elements of a system at a very micro level. The FSM is used to model an entire process at a macroscopic level. In regard to the formulation, the following differences can be pointed out.

1. The characterizing functions of the finite-state machines are thought of as decision tables, or Boolean functions. This is in connection with the type of symbols used. In the FSM the characterizing functions are extended to all sort of discrete functions, and in a sense they are treated similarly to the analytic functions. Also here, while the practical consequences are fundamental, no conceptual difference is involved. In the FSM moreover, because of the complexity of these functions, the characterizing functions of the machine are broken into several separated expressions, one pair for each state. Further subdivision in separate expressions will appear in the actual programing language.

2. In the finite-state machines, the input and output symbols are from a small alphabet; most commonly they are the binary symbols 0 and 1. This is because the finite-state machines are characterized in tabular form (or its equivalent), and with a large alphabet a table becomes impractical. In the FSM the symbols are sets of variables whose values are from a very large alphabet. Because the FSM is characterized in terms of functions, the size of the alphabet does not produce inconvenience. But in fact, while the different size of the alphabet produces large practical differences, it does not constitute a conceptual difference.

3. In the FSM there are the internal variables x_r . It is the presence of these variables that makes the FSM a practical model for all sorts of processes. However, the internal variables can be viewed either as further input symbols or as states of the machine. In either of these two cases the FSM would assume the formulation of the usual finite-state machine. In one case, we can say that the total states of the machine are partitioned in "process states" and internal variables. In the other case, we can say that the output symbols that are needed in the future are stored in the machine, rather than being represented as new input symbols. It is the flexibility given the user to choose each time which event is treated as a state, and which as an internal variable, that contributes to make the FSM a suitable model for the different types of processes. The possibility of exchanges between number of states and size of the alphabet in a finite-state machine was first shown by Shannon (1956).

Conventional finite-state machines have some memory capacity for holding the state. The FSM has the memory capacity extended to hold also internal variables. The interesting point is that the new finite-state machine (the FSM) becomes able to handle efficiently the processes without an outside memory and all the consequent traffic of data. Traffic which produces the unacceptable inefficiency of the Turing machines, and the accepted (but not desirable) overhead of conventional computers.

4. Conventionally, Moore and Mealy finite-state machines are distinguished in regard to the output production. In the FSM, because an entire programable network is at our disposal, a variety of output functions can be defined. In section 5.2.5, the language will be given for prescribing three types of output productions: state related, transition related, and driven outputs.

5. Finally, the most significant difference is in our use of the structural form. Based on the McCulloch-Pitts correspondence, we allow ourselves to consider generic structures that become specific structures in response to program words. This fact gives the possibility of viewing a sort of universal finite-state machine. The very interesting point is that, in spite of this universality, each specific behavior can be implemented with the desired efficiency, and not through a complex simulation.

6. The application of the notion of state made in our formulation deserves further comments. In section 3.1, the variety of levels at which the notion of state can be applied was discussed; but the application of states can be diversified also because of the different objects to which they have reference.

A well-known application of states is in the role of generalized coordinates of a system: the state variables. Here we are not interested in the detailed consideration of the values of such variables, a consideration that belongs more to the analysis than to the synthesis of a process; here, this role is performed by the process variables x_r . Our states correspond more to the phases of the process as visualized by the user. In psychological terms they tend to correspond to the states of mind of the user; in analytical terms, obviously, they can always be viewed as partitions of the set of values of generalized variables.

The usefulness of our definition of state becomes apparent in the construction of a program. The modularity offered by these states helps the assembling of complex processes by interconnecting and modifying different parts. These states provide an easy understanding of the grammar of the FSM; a state is a temporary choice of the four fundamental ingredients: new input prescription I , data transformation F , transition function T , and routing R . Besides, these states can always be used for modeling the different past histories, as in conventional finite-state machines.

3.4.2. Comments in respect to Turing machines

Turing showed the power of the symbiosis of a finite-state machine with an unlimited scannable tape. This symbiosis is the essence of all Turing machines, regardless of the various features that they might have. In the CPL automaton (Fig.11), the programable network constitutes a finite-state machine, the finite page memory plus additional outside storages that might be utilized through lines ξ and ζ are equivalent to an unlimited tape; the information read by the finite-state part is a page; the successive pages scanned are determined by the finite-state machine by means of the routing R . In this generalized sense the CPL automaton is a Turing machine.

Let us now comment on some peculiar features of the CPL automaton in respect to conventional Turing machines.

1. The portion of data that is read at each cycle (or move) of the CPL automaton is a page. A page can be considered as a tape symbol belonging to an extremely large alphabet, or as an elementary volume of a multi-dimensional tape (see Arbib 1969). It is well known that in general the size of the alphabet and the dimensionality of the tape allow an increase of efficiency in a Turing machine. In the CPL automaton, moreover, it is possible to relate in an intuitive way the structure of the pages to the data structures of the processes to be executed; and the overall result is that the work of the machine appears simple to a user who is familiar with those processes.
2. It is well known that a large number of states permits a more efficient work in a Turing machine; but, at the same time, it makes more cumbersome the description of the mechanization of the machine. In the CPL automaton, the number of states can be as large as required, and these states are introduced in correspondence to the mental states through which the user goes in thinking of the processes. The result, again, is that the work of the machine can be made efficient and simple to a user familiar with those processes.
3. A specific Turing machine is thought of as a definite structure, with definite quintuplets (symbol read, present state, symbol written, next state, move) that describe the performance of the finite-state part. A universal Turing machine has, in addition, a program written in a section of the tape, such as to instruct the universal Turing machine to simulate a specific Turing machine. It is precisely this simulation that makes the universal Turing machines unusable even at the theoretical level (the importance of the universal Turing machines is in their existence, not in their use, for instance, as theoretical models).

The CPL automaton, because it has the capability to store a program, is equivalent to a universal Turing machine. But its unusual characteristic

is that it does not simulate a specific machine, rather it becomes a specific machine, by means of the programmable network PN, and the structuring of the page memory. This is meant in the sense that the user, in each moment, sees only one definite, specific structure, the one that has been described by the quadruplet of the past and present states; at different moments, the structure may be different. One could say that the universality is obtained not by initially writing a program on the tape, but by initially assigning different quadruplets to the finite-state part.

The consequence, obviously, is that in this case we can optimize the efficiency for each different process.

4. The increase of difficulty in managing a Turing machine when the number of symbols and states increases, as mentioned in (1) and (2) above, explains why efficient Turing machines never have been considered. On the contrary, it has been a continuous challenge to search for Turing machines with fewer and fewer symbols and states (see Minsky 1967).

In the CPL automaton we can see a Turing machine where the efficiency is the prime interest. This efficiency is not paid for with a complexity in the "use" of the machine, because the machine is "constructed" as and only insofar as is necessary for each specific process. Certainly the substratum on which the machine is constructed (the PN and the memory) is much more complex than the substratum of a conventional Turing machine (a finite control and a linear array of tape squares). But this is of no concern for the user, because the substratum, be it simple or complex, is given. Moreover, in the CPL automaton, several different machines can easily coexist in a concurrent or independent work.

3.4.3 Considerations in regard to cellular spaces

Cellular spaces (briefly described in section 2.4) are indeed a fascinating field; to mention some of their potentials, they allow universality in both computation and construction; they seem to promise a mathematical understanding of certain aspects of biological structures; they seem to offer a coherent guideline for the developing technology of large-scale integration. However, to date, practical utilizations are quite behind the potentialities, either for modeling neural systems, or for describing parallel computations,

or for designing large integrated circuits. It might be that some crucial characteristics are still difficult to be handled in the cellular spaces so far considered.

The substratum defined in this chapter in some respects bears similarities with cellular spaces, and in other respects it uses different approaches. For this reason, some considerations in regards to both are discussed in this section.

Biological as well as psychological structures possess the capability of self-development from a genetic substratum; the new structures then constitute a new substratum for further development. The engineering of integrated circuits has the capability of adapting the design in accordance with an unlimited variety of practical considerations. Cellular spaces are mathematical structures that for necessity of definition and treatment need a high degree of uniformity and genetic simplicity. In cellular spaces, even with an amazingly simple mathematical substratum, universality in computation and construction can be theoretically obtained. But these outcomes, in general, have characteristics that render cellular spaces unsuitable for practical utilization. If practical considerations are imposed, the necessary mathematical homogeneity and simplicity are lost. The consideration of several hierarchical layers, each genetically based on a lower layer and exhibiting higher level characteristics might offer a solution to the problem; but no treatment of such an approach has been yet developed.

The substratum for our abstract machines is composed of two regions with different specialization. One region, the programable network or PN is specialized for operational structures; the other region, the page memory or PM, is specialized for data structures. This approach gives an initial advantage in respect to cellular spaces without going as far as to duplicate the rigidity of configuration of conventional computers.

In attempting practical implementations of cellular spaces there is always to be considered a compromise between the sophistication of the space and the loss of efficiency because of poor utilization of such an investment. In our substratum the solution is taken of using a limited, small region with a very high sophistication (the PN rich in connectivity and functional capabilities), and multiplying this region, in a virtual fashion by means of pages that reside in a much simpler substratum (the PM).

The page memory here is not a "random" piling of data, but is an organized tessellation of replicas of PNs, typically each one different from the others. When these replicas (the pages) are in the memory, they are, so to speak, in a dormant status; when they are in PN, they are in full active status. In this case, the compromise to be considered is between the extension of PN and the serialization of the execution. The utilization of the PN region can be made very efficient because of both its sophistication and its limited extension.

One of the difficulties in the theory of cellular spaces is that of communication among remote regions. The Holland's path construction is an example of the ingenious and complex provisions that have to be taken. In our substratum the circulation of the pages, in conjunction with the auxiliary storage H, provides a general means of exchanging data among different regions of the substratum. In addition, the routing function can be used as a fast mail system for sending information elsewhere.

The most complex task required by cellular spaces is programing. The desired activities have to be modeled in terms of the characteristics of the available space. The works made in regard to the von Neumann cellular automata and the Holland machines are examples. In the CPL automaton, viewed as a tessellation space, we can have easily different characteristics in different regions and these characteristics and regions can vary in time. Unlike the Yamada treatments, where these characteristics are dealt with as mathematical properties, here the user visualizes the properties in terms of characteristics of abstract machines. An abstract machine, when devised by the user for a particular task (for which he concerns) does not appear complex, psychologically speaking. In a sense, we may perhaps say that the approach of using a substratum that can be characterized in terms of abstract machines is in between those of cellular spaces and special purpose machines.

3.4.4 Considerations in regard to formal languages

Formal language theory defines a language as a set of strings of symbols over a finite alphabet. Such a broad definition covers natural languages, programing languages, and certain mathematical systems studied in automata theory. The approach taken in formal languages also clarifies how a language

makes it possible to express infinite information (the enumerable infinite set of strings) with finite means (the finite alphabet and grammar).

Our abstract machines, because they form an organized system of expression, constitute a formal language in a broad sense. But they are multi-dimensional objects, as opposed to linear strings of symbols, and they are governed by a grammar that also involves "imagery" features. As such they cannot yet be framed in available theoretical treatments (see Book 1973); this fact, however, does not impair the validity of using such a language. Moreover, we are using a very simple type of abstract machine that is readily expressible in the form of strings of symbols. In the absence of devices that could accept directly state diagrams sketched by the user, we are obliged, at the end, to express the abstract machines in the form of strings of symbols.

In section 2.1.4, the two main symbolic systems delineated by psychologists in the study of mental processes were reviewed: the verbal structure system and the imagery system. In this context, we can say that conventional formal languages are a way of expression that utilizes exclusively the phrase-structure symbolic system, and abstract machines are a way of expression that utilizes abundantly the imagery system. When a user is developing a program in the form of an abstract machine as illustrated in chapters 5 and 6, he uses both the verbal and the imagery systems as appropriate. When that program is given to a computer that can accept only strings of symbols, the abstract machine is expressed in terms of a formal language of phrase structure.

The interesting point is that certain processes might be more easily conceived and described in terms of abstract machines than by means of a phrase-structure language (compare with the von Neumann quotations on pages 60-61 in the previous chapter). In such cases, it is advantageous for a human user to work initially with abstract machines, for which he can take full advantage of imagery and geometrical intuition. These abstract machines should obviously constitute a formal system, that is they should form a "language" in the broad sense. Then, the abstract machines are re-expressed in the form of a string of symbols in a formal language, in order to be acquired by a digital machine.

The abstract machines formulated here can be expressed very easily in a formal phrase-structure language. The elements of these machines are always

the same (I, F, T, R), regardless of the process they represent. The grammar that manages these elements is formally simple, in spite of the tremendous flexibility that it offers for modeling processes of different natures. All the graphical means employed have a very direct representation in the form of symbols in a string, governed by a simple syntax. As an example, program 4215 expressed as a state diagram in Fig. 48 of chapter 6 is shown here in the form of strings of hypothetical phrase-structure language. Five FSMs can be distinguished, all engaged concurrently on the same task. The items constituting the FSMs are indicated sequentially in the order I, F, T, R, and the symbol ";" is used for missing items, except at the end of the state description. Each state is delimited with square brackets; the states are ordered following their label number, which thus does not need to be indicated. The unit that in spoken languages is the sentence, here is the state. The symbol # is used to delimit each FSM.

$$\begin{aligned}
 &\#4215[I_0F_0T_0(1,2)][I_1F_1T_1(0,5)][I_2F_1T_2(3,4)R_2][I_3F_0(0)][I_4F_4(0)] \\
 &\quad [I_5F_5T_5(6,13)R_2][I_6F_6T_6(7,9)][I_7F_7T_7(8,11)][I_8F_1(6)R_8] \\
 &\quad [;F_9T_9(10,13)][I_{10}F_{10}T_{10}(6,11)][;F_{11}][;F_{12};R_{12}][I_{13}F_0T_5(13,14)] \\
 &\quad [I_{14}F_0T_{14}(2,13)]\# \qquad \qquad \qquad \#1110[IF(0)R]\# \\
 &\#1112[;F_0T_0(0,1)][I_1F_1T_1(2,1)][;F_2T_2(0,3)][;F_3T_3(0,4)][I_4F_4(6/0)] \\
 &\quad [;F_5T_5(6/5,5)][;F_6;R_6]\# \\
 &\#1113[;F_0][I_1F_1;R_1][I_2F_2T_2(3,2)][I_3F_3][;F_4(0)]\# \\
 &\#1114[I_0F_0T_0(1,2)R_0][I_1F_1T_1(3,0)][I_2F_2T_2(0,3)][I_3F_3(0)R_3]\#
 \end{aligned} \tag{3.6}$$

It should be clear that Fig. 48 is not a graphical representation of strings of symbols that describes some procedure for implementing a process, rather it is the chosen model of the process; and expression (3.6) is the representation of that model in the form of strings of symbols. This reversed direction in the translation of the process description is a fundamental point.

From the analyses carried out in chapter 6, it appears, at the procedural level at least, that a formal phrase-structure description of abstract

machines of a proper type (machines that may represent processes of different types) is simpler than the formal description of the different types of processes directly in phrase-structure form. In the first case, the phrase-structure language has to deal always with the same structure (that of the formal abstract machine); in the second case, it has to deal with the variety of structures of the different processes. The abstract machines do not pose visible formal problems to the user because they are the reflection of what he has in mind.

In terms of the pictorial representation of Fig. 1, here we consider a point B (the abstract machines) that is very close to a point A, and a point C (the codes given to the computer) that is directly derived from B, as will be discussed in more detail in the next chapter.

Chapter 4

The Isomorphic Implementable Substratum

In chapter 3, a symbolic substratum was introduced for modeling (and thus describing) processes in the form of "constructs" (abstract machines) that are visualized by the user. The formulation of that substratum was implicitly guided by the realizability of an isomorphic physical substratum that could implement those "constructs". This chapter describes and discusses such a physical substratum.

4.1 GENERAL STRUCTURE

The symbolic substratum is organized in a structure called the Circulating Page Loose (CPL) automaton, as represented in Fig. 11. Such a structure is also the basic structure of the isomorphic physical substratum. In fact, it was the known realizability of such a structure that suggested its adoption in the symbolic substratum. Because of its identity with the CPL automaton, the physical substratum will be referred to as the Circulating Page Loose (CPL) system. In essence, this structure consists of a programable operating network and a programable memory through which pages of data circulate. However, now we have to consider certain details of the real world that were advantageously ignored in a symbolic context.

4.1.1 Interface with the environment

In the real world, we cannot expect the environment to be at our beck and call. In fact, the environment is composed of peripheral devices, usually with low speed, with their own timing, and without any knowledge of what is going on inside a computer. Therefore, buffers and means for selecting and controlling the peripherals will be necessary. Accordingly, we introduce

two auxiliary units (Fig. 13): an "assembler" between the output of the memory and of the environment and the input of the programmable network PN ; and a "packer" between the output of PN and the input of the memory and of the environment.

These units provide for the interface between the system and the environment, as well as that between PN and the memory. The assembler is assigned the implementation of the input prescription I, and the packer that of the routings R. Every time the assembler receives a page from the memory, it calls for specific new input data U from the environment, responsive to the information in I of the present state of the page. It assembles the page into an array of registers Ω_a , called the assembler page array, which is a replica of the registers in PN. In Ω_a , the page assumes again the open form that it had in PN at the previous cycle. As soon as the assembling of the page in Ω_a is completed, and the programmable network PN is free, the page in Ω_a is transferred in parallel into PN. At this point, the data structure of the page becomes part of the operational structures of the F in PN, and the data transformations described in the present state of the page are executed.

At the end of the operations in PN, the page is transferred in parallel into an array of registers Ω_p in the packer. Here the routing prescriptions are implemented. Responsive to information in the R of the present state of the page, outside devices are activated, and data are transferred. At the same time, the variables x_i in the page are packed for transmission into the storage medium of the memory. During this transmission, the movement of data within the page, prescribed by routing, is implemented also.

Note that outside devices may be not ready to give or acquire data when a page is transiting through the assembler or packer. A variety of solutions can be adopted for these cases. On one extreme, the FSM can have an initial state in which the page circulates through path β in order to acquire all the data necessary to start the processing, and an output state in which the page again circulates through path β in order to deliver all the results to the environment. In the same time that these peripheral operations are accomplished, other pages follow path α for the execution of their processing in PN. On another extreme, sufficient buffer storage and peripheral control can be added in order to make the work of the pages completely undisturbed by the environment timing.

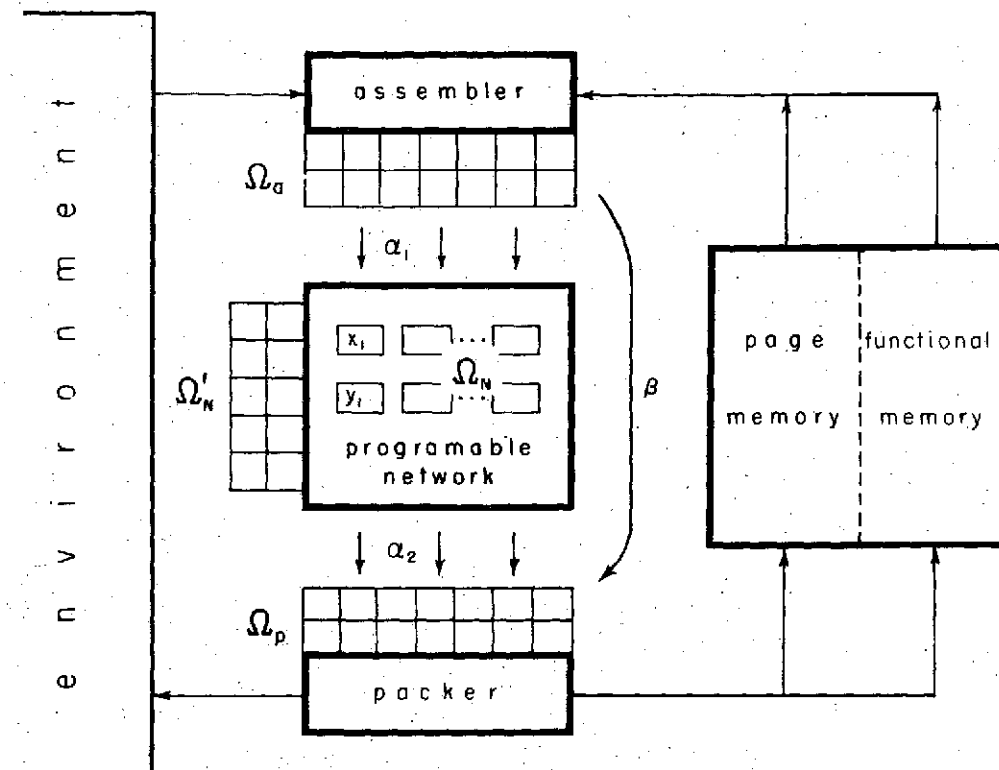


Fig. 13 - The basic frame of the CPL system

The introduction of the assembler and the packer, however, does not make the system different from the CPL automaton. When the user is concerned with the processes, assembler and packer are invisible. Only when he is concerned with details of input and output does the presence of the assembler and packer comes into light.

4.1.2 Program allocation

When a process consists of a single page of data and a very concise FSM, the description of the FSM (that is, the content of storage P in Fig. 10) can be very well part of the circulating page. The quadruplets are acquired from the environment in the initial state; during processing, the quadruplet of the present state is brought to an active position; at the end of the process, all the FSM description disappears by not recirculating.

In general, an FSM description has a size exceeding what is appropriate for a circulating page. Moreover, often an FSM is implemented by many pages.

For these reasons, separate storage is provided for holding the quadruplets of all the FSM in use. The pages instead, carry a key word that contains the label of the FSM to which the page belongs, and the label of the present state in that FSM. When a page arrives into the assembler, a specific quadruplet is acquired from that storage, in response to the content of the key, and is added to the page in Ω_a .

Component I of the quadruplet is utilized by the assembler for providing the new input U, and no longer follows the page. Components F and T are utilized by PN, and do not follow the page in the packer. Component R is utilized by the packer, and does not follow the page in the memory. As a consequence of function T performed in PN, a new key will in general substitute for the old one in the page; this key will indicate the next state for the page, and optionally also a new FSM. The auxiliary page array Ω'_N permits also the transmission or interchange of program components among different pages.

Because the specific means that are implemented for giving the pages a quadruplet are not visible to the user, there is no indication of them in Fig. 13.

4.1.3 The automatic flow of data

In the CPL automaton, the user can develop abstract machines easily because an active substratum is available that produces, recirculates, and interrelates pluralities of pages. The user is not concerned with how all this happens, he knows only the spatiotemporal frame in which the activity occurs (Fig. 11), and the means for building and affecting that activity (the configurations of PN and of the memory). A similar situation should be maintained in the physical substratum. Techniques invisible to the user should materialize this activity as an inherent characteristic of the system.

The actual implementation of the automatic circulation derives from the general structure. One page at a time is prepared in the assembler by receiving old data from the memory, new data from the environment, and a state description from the program storage. The programmable network and the packer perform their operations as soon as they receive a page. The memory builds pages, or data structures, as soon as it receives them from the packer.

The basic page transfers can be described with the use of register

transfer notations (Bartee et al 1962) by the expressions

$$\begin{aligned}
 t_1 \alpha_1 \Omega_a + t_2 \gamma F_1(\Omega_N, \Omega'_N) + t_3 \delta_1 \Omega'_N &\rightarrow \Omega_N \\
 t_2 \gamma F_2(\Omega_N, \Omega'_N) + t_4 \delta_2 \Omega_N &\rightarrow \Omega'_N \\
 t_5 \alpha_2 \Omega_N + t_6 \beta \Omega_a &\rightarrow \Omega_p
 \end{aligned} \tag{4.1}$$

where F_1 and F_2 are functions executed by the programable network; t_1, t_2, \dots are Boolean time functions produced by a control system; and α, β, \dots are Boolean conditional coefficients with value, meaning, and constraints as shown below.

α_1	α_2	β	γ	δ_1	δ_2	Condition
1	ϕ	ϕ	o	o	ϕ	acquisition of a new page
ϕ	1	o	o	ϕ	ϕ	recirculation of a page
ϕ	o	1	ϕ	ϕ	ϕ	recirculation of a page bypassing PN
o	o	ϕ	1	o	ϕ	processing of a page
o	ϕ	ϕ	o	1	ϕ	acquisition from storage
ϕ	ϕ	ϕ	ϕ	ϕ	1	storage of a page or data

The system can process in sequence all the pages through the paths α_1 and α_2 ; it can continuously process a single page, condition γ ; it can input and output data without involving PN, following path β ; it can buffer a page for a certain time in the auxiliary page array Ω'_N through the transfers δ_2 ; it can produce a new page in array Ω'_N during processing (combination of paths δ_1, δ_2 and γ) for the execution of a subtask; it can introduce the new page into circulation through transfer δ_1 .

The registers of arrays Ω_a and Ω_p have one-to-one correspondence with the registers Ω_N embedded in the programable network; the packer transfers the data in Ω_p into a page for the memory in a given order; the same order is used by the assembler to allocate the data of a page into Ω_a . In this way each variable of a process always goes into the same register of PN,

during the circulation of the page, if not otherwise prescribed by the program. These structural features of the physical system implement automatically the virtual multiplication of PN (by means of pages) that was implied in the symbolic CPL automaton.

The circulation of pages, which is always present if there is any activity in the system, implies a storage with properties appropriate for receiving, holding, and delivering pages of different characteristics. Such storage, because it is always present, is specifically indicated in Fig. 13 as page memory.

But different processes have different data structures, and even the pages may alternate periods of circulation with periods of rest or of different activity. For these reasons, a storage is also necessary that can assume different properties and different partitions in accordance to the different data structures needed. Because the extent of these storages and their input-output disciplines are not established in advance, but are determined at each moment by the processes themselves, the general term of functional memory is used in Fig. 13. Which functions are physically implemented in the memory depends upon the design; in chapter 5, specific designs will be referred to. Obviously, it will be convenient to have always the function of addressable storage, which in our context can be described as follows: the access to an old variable x_b (that has been routed to some data structure in the functional memory) by means of a variable x_a in a page.

In general terms, a specific amount of storage medium is given to the physical system; then a control is added such as to partition that medium in several regions that implement different disciplines and that can be independently accessed by the packer and assembler. The orders for this control are given by the FSMs in the form of routings to particular control storages z_r (item vii in the symbolic formulation of section 3.2.1). In other words, it is a hardware implementation of what usually is implemented in software.

4.2 IMPLEMENTATION

All the parts of a physical substratum that is isomorphic to the symbolic substratum defined in chapter 3 are implementable with the present technology of integrated circuits. No particular difficulty was encountered

in implementing a first CPL machine even with the components available in 1968 (section 4.4). Thus the real issues of interest are performance and analysis of design. Documentation on performance is given in chapter 6. Design involves consideration of a variety of contingent requirements that are out of the scope of this introductory report. Therefore only brief comments will be made on the more unconventional aspects of the system, with an attempt to distinguish what is intrinsic to the approach from what are contingent factors.

4.2.1 The programable network

Networks of operating elements have recurrently attracted the attention of people, in both a theoretical and a practical context. In section 2.4 studies were reviewed in the context of cellular spaces and array computers, and in section 2.5.4 in the neurologic context. First studies on networks of logical elements appear in Burks and Wriah (1953). An extended survey of networks oriented to microcircuit implementation can be seen in Minnick (1967). Examples of subsequent contributions are in Meo (1968), Sheldon (1972), Jump and Fritshe (1972), and Maruoka and Honda (1973). Recently, networks are studied in the context of distributed computers.

In this broad activity related to the notion of network, a few points relevant to our context are discussed.

Networks at the system level do not permit an intimate collaboration of the available resources, and they create traffic problems with a consequent high overhead. Networks as arrays of elements have two basic disadvantages: (1) to perform a specific function, a complex program or control is needed that typically has no relation with the forms in which the functions are visualized by the user; and (2) arrays require many more elements than would be needed in especially designed structures.

A notion that appears of primary concern in all these works is regularity: geometric regularity, functional regularity, time regularity, etc. Probably in many cases the emphasis on regularity derives from the comfort of an elegant mathematical treatment more than from pragmatic necessity.

The programable network considered here does not start from regularity of structure. It is oriented to the implementation of the different structures that the user conceives for different problems. In a sense, it has an approach similar to that of patchboards in analog computers; the structures

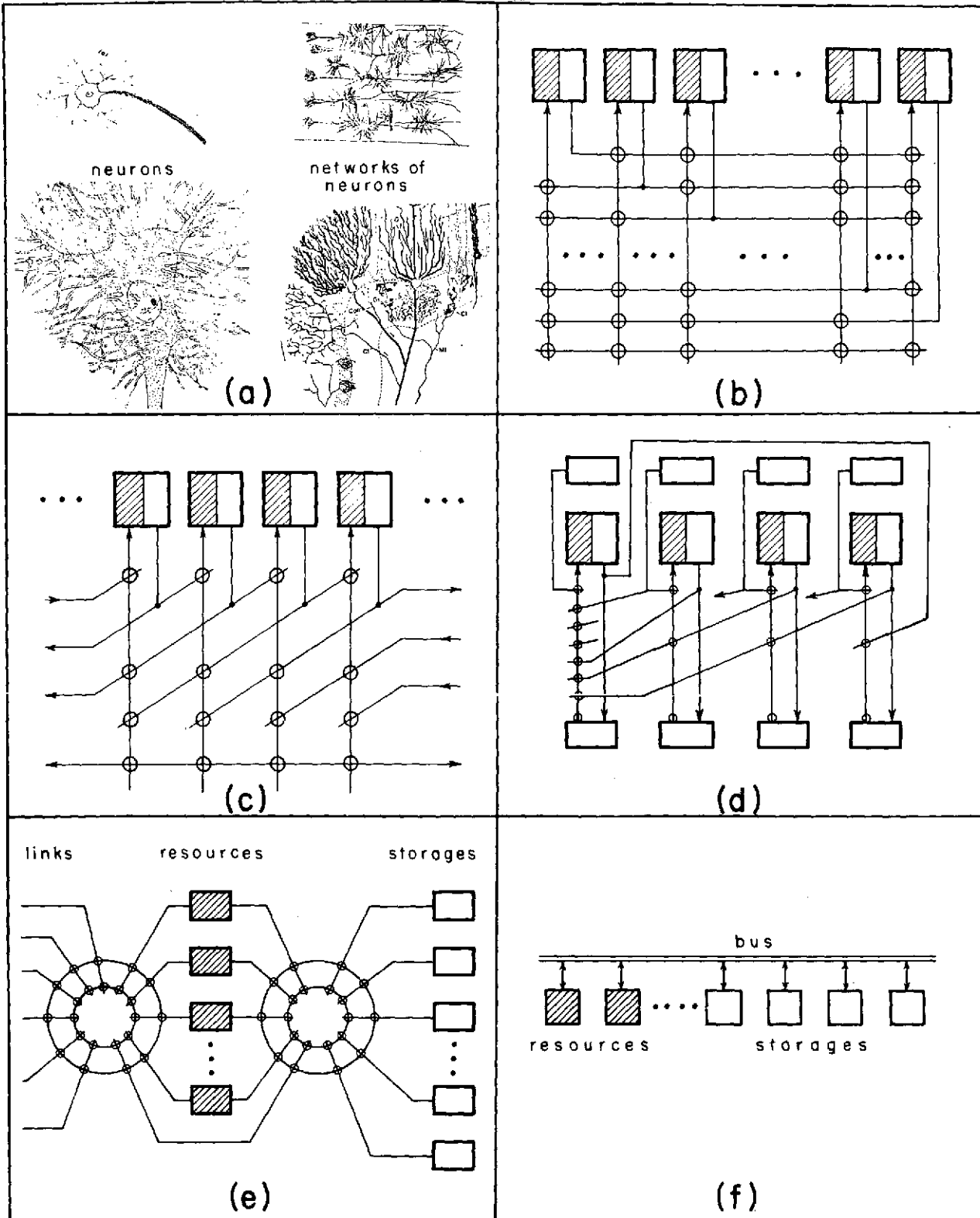


Fig. 14 - Different types of connectivity

assumed in a programmable analog computer are those chosen by the user as the model of the process to be executed.

It is not difficult to implement in hardware all sorts of complex operations comprising many variables (special purpose processors are such implementations). The real question is how to form a language that allows the implementations of all those different designs, from a standard programmable substratum, by the part of nonengineer users. For instance, in a fixed plus variable structure computer (Estrin et al. 1963) it was expected that programing "should be done by a team consisting of a numerical analyst, a programmer, and a computer design engineer". The solution that has been found here is to use recursively the approach discussed at page 62, and already used for the general structure of the CPL system. The hardware of PN forms a programmable substratum isomorphic to a symbolic substratum where the user can describe structures for data transformations that he derives from his images of the data transformations. A variety of modes are available for specifying (sometimes in steps) these structures, in order to conform to their different characters.

In this way, the mapping (3.5) of section 3.2.2 can be obtained by application of rules to elements of a given alphabet, with the alphabet and the rules having a meaning to the user. The more complete the isomorphism is, the more the following situation can be approached: everything that is expressible in the language can be implemented in the network, and all the possible configurations of the network (the hardware tricks) have their logical expression in the language.

In this approach, programability has such depth and extension that no processing characteristic can be attributed to the physical structure per se when isolated from the program. We can recognize only two principal characteristics in the substratum: functionality and connectivity.

Ideally, the functions available to the various elements should include the functions that are recognized in the mental processes. Psychologists make their research in this direction; see for instance Piaget (1950). More fruitful results for the present context can come by making such analyses with particular attention to computer use. A preliminary experiment is described in section 5.1.3. In the absence of such analyses, the few arithmetic and logical functions can be augmented with others related to data manipulation. Fundamental in the functionality of a network are functions in the form of look-up tables.

Connectivity is no less important than functionality for approaching isomorphism with mental constructs. A set of different types of connectivities is depicted in Fig. 14. In (a) is the connectivity found in biologic neural systems, where functional elements (neurons) have tens of thousands of possible connections. Very probably, this approach is not suitable for man-made devices. What can be derived from (a) in a technologic context is the total connectivity represented in (b). Each functional element has an output line that is connectable to the input of all the other elements. Note that we do not separate storages from functional resources; here, an element is at the same time a storage of a quantity and a functional resource.

Approach (b) has the practical inconvenience that it is not expandable; the total number of elements should be known when constructing the input selector of each element. The connectivity represented in (c) is not total but permits an unlimited expansion.

Obviously, one is interested in the minimum connectivity for a certain degree of performance. Probably, this can be obtained only by means of non-uniformity. The connectivity represented in (d) has a preferred element that has connection with all the other elements, and the remainder elements have connection with one neighbor, with a replica of itself, and with a source.

Computers that search for performance without economy in connectivity use approaches of the type represented in (e). A significant overhead due to traffic problems is inherent in this solution. Computers that search for maximum economy use the connectivity represented in (f).

There is an interesting rationale in a completely programable network. With a linear increase of hardware elements, either as number of elements or number of connections, the operating configurations that are possible increase with an exponential function, and the length of the expressions for describing them increases only logarithmically. For given criteria of performance, an optimum complexity can be expected for a PN.

The various solutions that can be adopted for the programable network have known technical implementations. Because of the isomorphism between the physical and the symbolic substrata, the description of those solutions can be made either in the hardware domain or in the symbolic domain. The latter is chosen here, and symbolic descriptions are given in chapter 5.

4.2.2 The programable memory

A variety of media for storing digital information are well known and established, and new media are continuously in development. These media can be grouped in three types, according to their inherent mode of inserting and extracting data: sequential (e.g. magnetic bubbles, tapes); cyclic (e.g. drums, delay lines); and random access (e.g. core memory).

From any one of these media, memory with any input-output discipline can be formed. Early computers implemented random access memories with delay lines and drums. Sequential or cyclic memories can be implemented with core memories whose addresses are connected to a counter. Content addressable memories can be implemented with serial, cyclic, and random access media. Techniques for implementing the different types of memory from the different media are well known.

In the CPL system, the discipline for data storage varies in accordance to the characteristics of each process, thus a programable control has to be used in the memory. In this condition, a large variety of storage media can be used as well. The particular characteristics of each storage medium will make the control implement some modes of storing directly, and other modes less so. An addressable storage offers a more uniform complexity in implementing different disciplines of data storage.

In regard to the page memory, fixed or variable formats can be used. Auxiliary signals or flags are needed for delimiting the pages and their parts. The printed pages with their punctuation marks offer an interesting example of how information can be related to the structure of a text. Techniques for implementing circulating pages were previously described (Schaffner 1966).

The term functional memory as used here refers to the organization of the storage medium for obtaining a data acquisition generalized to a function. Examples are: augmentation of the present content with a new datum, as in an accumulator; increment by one at each access performed; and substitution with the largest of the present content and the new datum. Obviously, among the functions of the functional memory, there will be always the selective transfer (read/write) as in conventional random access memories. Symbolic description of functional storage is in chapter 5.

4.3 DISCUSSION

In this section we look at the physical substratum as a computer and analyze its structure and characteristics as compared to those of various computers that have been built or proposed.

4.3.1 Pipelining

The architecture represented in Fig. 13 clearly has a pipeline organization. While the assembler is preparing a page, PN is processing another page, and the packer is routing a further page. While the several configurations of F are succeeding in PN, input and output buses give and take data to and from the assembler and packer, and give a stream of state descriptions to assembler, PN, and packer.

The overlapping of the processor and memory operations (Buchholz 1962) is here intrinsic to the basic structure. In conventional computers the efficiency of pipelining is strongly dependent on the presence of a stream of similar tasks (Graham 1970; Ramarmoorthy 1972); here all portions of the processes are framed in the standard form of the FSM. Packer and assembler have independent data channels, and their operation times can be statistically matched by providing sufficient buffer storage with a first-input-first-output discipline, thus approaching a full time operation in PN (cf. Cotten 1969).

A characterization of computers in terms of data and instruction streams has been suggested by Flynn (1972). Here the appropriate characterization is in terms of FSM and page streams; we do not deal with operands and instructions, but with data structures (the pages) and operating structures (the FSMs). The relevant difference is not in the size but in the level. A page is not an amorphous segment of data. It is a self-sufficient package of information, in the sense that it contains all that is necessary to perform a portion of processing (a state) in PN. It may correspond to a job. The key accounts for the program, new input data will be found ready in PN, and other broadcast or exchanged data will be found also in PN (the X' in the auxiliary page array Ω'). Even when the page is sleeping in the memory, reduced to the minimum $X + \text{key}$, it remains an organized set of variables because it has (in a potential form) the same spatial configuration it had, or will have, in PN for matching the operational configurations. Each word describing the FSMs

has a much higher level than that of conventional instructions because it refers to a programable network rather than to a single processor.

4.3.2 Addressing

Computers are articulated on two basic parts: a computing machine - the processor - , and a storage of data - the memory. To make these parts work meaningfully, an addressing function is necessary. Conventional computers feed data into the memory randomly, and thus they need to assign them an address, and add to the processor an address-manipulation part.

In the CPL system, the processor takes the form of a programable network that is also the storage of the data it uses. When an extension of this storage is needed, replicas of it (the pages) are formed in a larger storage (the memory). Moreover, the organization of this larger storage is adapted in time to conform to the required movement of data in the various processes. Under these conditions, the addressing function, in the conventional sense, disappears. In its place there is the structure of each page and the grouping of the pages.

The structuring of the pages is automatic. At each moment/time, a page has the structure that was given to it by the last union with an operating configuration in PN, and by the routings. Each operating configuration and routing in the FSMs is prepared by the user in accordance to the page structures that are to be met. The several state diagrams in chapter 6 illustrate this interplay between FSMs and pages.

The grouping of the pages is a consequence of specific actions of the FSMs. Pages are created by the FSMs; new pages can be inserted at any point of an existing array of pages; pages can be deleted. Arrays of pages can be scanned in different ways by means of commands in the FSMs. Examples of isolated pages, one-, two-, and three-dimensional arrays are illustrated in chapter 6.

In conclusion, the main addressing function is accomplished without the use of individual addresses, but by means of pages as dynamic movable storages -- dynamic in the sense that their structure changes in accordance to the structure of the information they hold; movable in the sense that it is not a physical location where the information resides, but a structure that goes from one medium to another (the PN and the structurable memory). Also in conventional high-level programming languages there is no use of addresses,

but simply because the task is shifted to the compiler. Internal addresses and their manipulation account for a large part of the memory and of the overhead of conventional computers.

Often in a process it is simpler to address a specific nonlocal datum whenever necessary, rather than to establish a special data structure such that the datum appears spontaneously at the needed time and place. For these cases, the CPL system has a functional memory available, in which the function of addressed storage can also be performed.

Sometimes in a process there is a sorting of data controlled by rules or information not available in advance, for which no data structure can be prepared in the program; that is what is called a random process. But in all these cases the particular information that will determine the actual sorting of the data will necessarily appear as a variable x_r in the process. Therefore, that variable can be used as an expressly created address for accessing the needed data in the functional memory, or in some page structure. Evidently, among the functions available, there will be the one that uses a variable in the page for addressing other data structures.

We see that a programable substratum is capable of taking advantage from various addressing methods. Well-designed, special-purpose digital machines in general use few or no internal addresses; the structure of the machine provides for the needed flow of data. Similarly, a well conceived abstract machine (i.e., a program for the CPL system) will in general make little or no use of addressed storage.

In this light, the total addressing in a random form of conventional computers appears as the most onerous solution. It is the price paid for having full flexibility with a computer of rigid structure. In the CPL system, because structuring is permitted, much less price is paid for full flexibility. We do not need to handle addresses for all data; we provide addressing information only when there is a change of structure.

An interesting situation also arises with program addressing. In conventional computers, the main program uses names for addressing routines that are stored somewhere. In the CPL system, the FSMs use words such as the F, T, and R which are in themselves all the necessary information for accomplishing the tasks indicated with those words (see the mapping (3.5)). When an F or a T is attached to the p inputs of PN, an entire operating

structure is implemented, and the operands are already present in the structure. When an R is routed to the memory control, a new data structure is formed, and the data are already present in the storage medium.

4.3.3 Parallelism

Parallelism is one way for increasing the throughput of computers that is actively pursued both theoretically and practically. The classical scheme of parallel computers consists of an array of identical processors obeying a common stream of instructions. The performance of this scheme is heavily dependent on parallelism in the problems. Whereas for particular problems parallel computers can achieve a throughput which is orders of magnitude larger than that of conventional computers, for general problems they face a performance degradation that increases with the number of processors, due to the difficulty for the operating system to keep busy all of the processors, and to contentions in the access to the memory (cf. Chen 1971). Moreover, parallelism is generally not simply expressed in programming languages.

For these reasons attention is brought to reduced form of parallelism in the hardware and to the exploitation of the inherent parallelism in the computations (cf. Hobbs et al 1970; MAC 1970). Examples of studies toward a general understanding and modeling of parallelism in the processes are in Karp and Miller (1967), Slutz (1968), and Thomas (1971).

In the CPL system, high throughput is sought by means of the specialization of the hardware, both in the processor and in the memory. Parallelism is one of the specializations adopted in programming to the extent to which parallelism exists in the process to be executed. Two forms of parallelism are possible.

- (1) In the programmable network all the variables x_r can perform operations simultaneously and independently. Also different operations can be assigned to these variables. Each variable can deal with its own input u_r and auxiliary data x'_r .
- (2) An array of pages can perform the same FSM, thus implementing parallel processing in a virtual form. In this case the actual execution occurs serially, but from the programming viewpoint the array of pages can be treated as a parallel computer. Form (1) can coexist with form (2).

Interesting is the fact that questions of timing and interconnections

are solved in the same moment in which the user establishes the structure of the pages and their interplay with the FSMs. Examples of different degrees of parallelism are given in chapter 6.

4.3.4 Computer architecture

In the history of computers many designs have been proposed and several implemented (see Bell and Newell 1971). The intent in each case is to find the computer structure that optimizes given requirements. However, every time that an elaborated structure has been used, it turned out that the computer is appropriate for the objectives for which it was conceived, but is less so in other respects.

Present general purpose computers, in order to exhibit a more uniform response to the different classes of problems, adopt nonspecialized processors, random access memories, and shift to the software the burden of implementing specializations.

It is natural to ask whether an adaptable specialization could avoid the nonuniform response of the specially structured computers, and the inefficiency of present general purpose computers. There have been specific attempts in this direction, perhaps the most conscious are the fixed plus variable structure computer of Estrin (1960-63), a polymorphic data system (Porter 1960), and the distributed processor of Koczela (1968). However, neither of these or other suggestions have motivated the production of structurable computers.

The solution described in this report suggests a computer structurable at a very intimate level; and to such a degree that we can attempt to make the computer implement the structures conceived by the user in modeling the processes.

To visualize the structure of the CPL system as a computer, the PMS representation (Bell and Newell 1971) can be advantageously used. At the most global level, the CPL system appears composed (Fig. 15a) of a processor P connected to a memory M and an environment X through a transducer T, as any other computer. If the switches S and the partition of the memory into the various functions are made visible (Fig. 15b), we can visualize the pipeline structure. Note that the arrows indicate a continuous flow of data rather than random bi-directional traffic. If the controls K and the data transformation facilities D are also made visible (Fig. 15c), we can visualize the parallel, or in general, the specializable structure.

P = processor

M = memory

S = switch

K = control

D = data processing
facility

T = transducer

X = environment

— data flow

--- control flow

X — T — P — M (a)

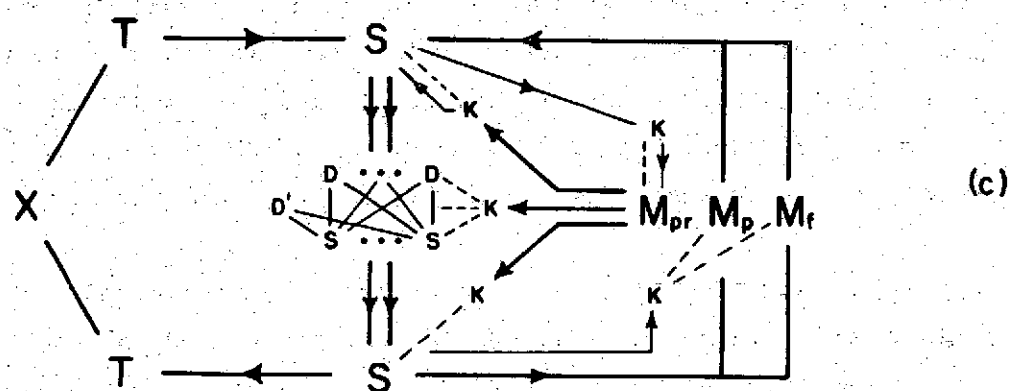
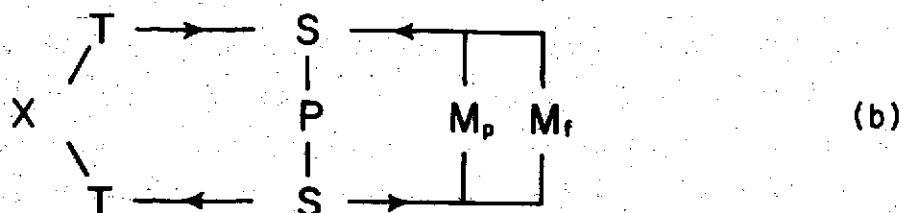


Fig. 15 - PMS representation of the CPL system

In Fig. 15c we can easily visualize the work of the CPL system. Two flows of data, one from the environment and one from the memory, merge into the programmable network PN. From PN, two data flow emerge, one directed to the environment and one back to the memory. Two sets of switches are basic to the work of the system: the connectivity of PN (the small s in the figure) which implements the operational structures; and the assembler and packer (the large S in the figure) which implement the movement of the data structures. The memory is partitioned in three regions: one, M_{pr} , for storing the FSMs descriptions; one, M_p , for holding the pages, which are virtual replicas of PN; and one, M_f , for storing particular data structures. The controls K at the programmable network, memory, assembler, and packer implement the dynamic structuring of the computer in accordance with the structure of the processes under execution.

One basic characteristic of this architecture is that most of the memory is not treated as a part separated from the computing machine (the processor), but rather as a storage of virtual replicas of the processor.

But perhaps the most significant diversity of a CPL computer is in the different use of the user's intelligence. A programable substratum is not enough for making a computer. Also necessary is a kind of compiler that is able to prepare the appropriate organizations of the substratum. As is well known, this is not an easy task; especially if the source program is in a phrase language (cf. Wineberg and Avizienis 1972).

But if the substratum has enough flexibility, we can ask the user to provide for the organization of the substratum. In particular, we can exploit the natural capability for images that all users have, bypassing a verbal description of the process. If the substratum permits a sufficiently flexible structuring, the user does not have the impression that he is dealing with a piece of hardware, but that he is producing representations that are no less rigorous, or less symbolic, or less elegant than the phrase structure ones.

There is one requirement, however. It is clear that human beings do not use the same structures for all types of problems. Therefore, the substratum should allow a variety of structures, including verbal structures. In these cases, some FSMs can act as on-run compilers.

A very detailed programability is favorable also for the economy. What in conventional computers are the different units, here are different structures implemented on the same common substratum. This brings a higher utilization rate of the hardware. The cost of the programability, because of the direct interpretation of words to form configurations, is significantly less than the total cost of the controls in separated units for performing the same tasks.

The organization of the computer activity in terms of pages makes the references to the memory less frequent and more predictable, for which economy can be derived in the implementation of the memory.

4.4 THE CPL 1 PROCESSOR

4.4.1 Factual information

In the early 1960s, efficient and economical real-time processing of radar signals was required at the Harvard College Observatory for the Radio Meteor Project, an astrophysical research program.* An effective solution was found in using specially designed hardware in conjunction with circulating digital words (Schaffner 1964). Several systems based on this concept were installed at the project radar station in Havana, Illinois (SAO 1966).

In December 1964, at the Smithsonian Astrophysical Observatory, a small demonstration was given of the instant implementation of different processors by changing a patchboard in a general digital structure. The analog approach of structuring the machine in accordance with a model of the process was there applied to a completely digital system.

Then, it was a natural step to replace the patchboard with program bits. In this instance, these bits were holes in a punch card. Also the first IBM computer was a Card Programmed Calculator (CPC). The divergence that followed for the CPL system, in retrospect, can be attributed to the following reasons: (1) the approach of organizing the machine after a model of the process, analog computer style, was retained, rather than discarded for a purely verbal description of the processes; (2) general frames already developed in automata theory were assumed for modeling both the processes and the hardware.

Subsequently, the development of the CPL 1 machine was initiated (Schaffner 1966). The first operation started at the end of 1968. During 1969 the machine was used for testing a variety of computations in the laboratory, and preparing the interface with the environment. In 1970 the CPL 1 processor was brought to the radar station of the Project in Havana, Illinois, where recording of faint meteors with different recognition strategies were made (see section 6.3). Then the equipment was brought to the Massachusetts Institute of Technology (see section 6.2).

Under the present contract, an analysis of the system was undertaken. The context of automata theory was joined with the consideration of studies of psychologists on mental processes, and the notion of isomorphism between a symbolic substratum and a physical one took shape. In the symbolic sub-

*Originally supported by NSF under Grant G-14699, then by NASA under contracts NASr-158 and NSR-09-015-033.

stratum the user describes his images of the processes; the physical substratum implements those images. This study constitutes the basis for the programming language of the CPL system.

The 1966 report was made before the construction of the machine, with many issues still to be understood and developed; the present report is made after use of the machine, and represents a first analysis of the approach taken.

4.4.2 Description

The CPL 1 machine has the architecture represented in Fig. 13. The programmable network PN has four variables x_r , four variables x'_r in Ω'_N , four new input data u_r , and a key word. The controllable traits (connections and operational characteristics that can be described by program) are 148. The functionality at the variables initially consisted of setting, resetting, complementing, shifting, transfer, summation, and subtraction. Then special functions were added as they became desirable. Many of these functions have a variety of controllable details. The connectivity is of the type indicated in Fig. 14d. The registers that hold the variables can be cascaded in order to form words of different length. The basic segment is 12 bits long.

The program is stored in a separate storage. The key word of each page acquires into the assembler a state description which is composed of four 4-bit words for the input prescription I, three 12-bit words for the function F, four 4-bit words for the function T, one 4-bit word for the routing prescription R, and a 12-bit code word. The bits representing I, F, T, and R are interpreted in different ways according to the content of the 12 bits of the code word. One of the most interesting aspects in designing the CPL 1 machine has been the realization of the possible utilization of the totality of the bit patterns in a meaningful language - the mapping (3.5). For this reason, simple 12-bit words can describe a full configuration of PN. In each page cycle, up to three different configurations can be implemented for F, and one configuration for T. All words W_F and W_T are implemented directly, and no microprogram processor is used. The FSM descriptions are written on punch cards; the format used is described in section 5.3.

The assembler has three page arrays Ω_a (Fig. 13) for matching the data rate of the memory with the processing times of the programmable network. Available inputs are analog signals, digital words, and numbers set on dials.

The packer has an output buffer to permit the recording of output data simultaneously with new processing on the part of the pages. Output peripherals are a magnetic tape recorder, a fast printer, and oscilloscopes. All variables x_r can be observed during their passage through the packer, in both analog and digital forms; this feature permits the display of the evolution of quantities of interest during computation. A diagnostic facility permits the damping of all variables at the exit of each cycle, or alternatively at each change of state. The reasons discussed in section 5.4.1 with regard to the ease of debugging programs apply also in the ease of checking the hardware - another consequence of the isomorphism between the symbolic and the physical substrata.

The memory is implemented with MOS devices. Part of it is used for the circulating pages, and part for particular data functions.

A separate supervisor unit receives information from a digital clock, operator push buttons, and a supervisor program; as a consequence of this information, the supervisor injects into circulation pages of different FSMs, at proper time and range intervals. In one program card of the supervisor it is possible to schedule for an hour of processing up to 15 FSMs in consecutive or periodical different arrangements, with the resolution of one second in time and one kilometer in range.

The entire equipment is contained in 22 printed circuit cards. In Fig. 16 the CPL processor and the supervisor unit are visible at the right of a rack of peripherals and the radar console.

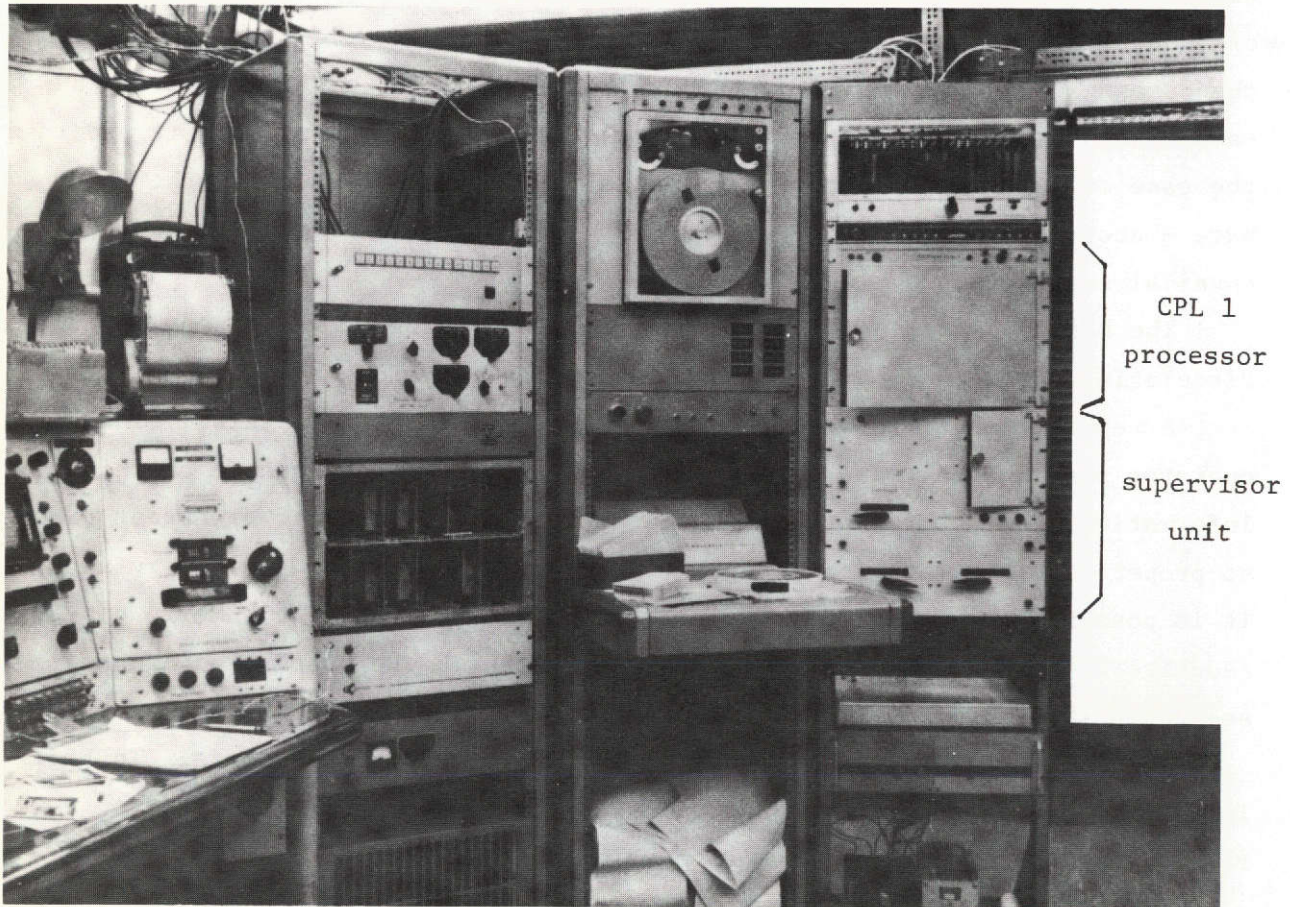


Fig. 16 - The CPL 1 machine at the M.I.T.'s weather radar

Chapter 5

The Programing Language

In chapter 3 a symbolic substratum for abstract machines was introduced; details that are not fundamental to the structure of the substratum, and are thus open to a variety of implementations, were not given at that time. In chapter 4 a physical substratum isomorphic with the symbolic substratum was described, again without details pertinent to particular implementations. In this chapter we give actual means for describing the abstract machines, and molding the physical substratum in accordance with them, including choices for those details. These details apply to both the symbolic and the physical substrata. A preliminary discussion on the role of a programing language begins the chapter, and a comparative discussion in reference to other programing languages concludes it.

5.1 INTRODUCTION

5.1.1 - The role of the programing language

Following a classification of Burkhardt (1965), programing languages for digital computers range from

- machine codes
- to - assembly languages
- procedural languages
- specification languages
- and - declarative languages

in accordance with the use made of interpretive or translating routines (compilers). Languages were also proposed that simply state the problems, without indicating the solution to be used (cf. Schlesinger and Sashkin 1967).

This extreme case however cannot be viewed simply as a programming language, but should be considered more as a system of solution finding.

It is a characteristic of conventional programming that a process is described in several forms: in the user language, possibly in the intermediate languages of the compiler, in an assembly language, and finally in the actual binary codes of the computer. In each one of these forms the process is completely described; each form is obtained from the previous one by translation. All these different languages can be viewed as formal systems of verbal structure (although the machine codes are at the limit of such a view).

Programming procedures for analog computers, instead, need a completely different approach. These procedures involve such a diversity of activities and gadgetries that they can be hardly viewed as a language. However, because they convey definite information, they constitute an actual language in the broad sense. In such a variety of contexts, it is not possible to define the role of programming languages in general, we can instead clarify the role of the programming language in our context, and confront it with the typical ones in other languages.

In chapter 3 we developed a method for modeling processes. Modeling relates to the mental structures employed by the user during the conception of a process; it does not relate, in general, to an actual computer. In chapter 3, however, we were careful to frame the modeling in a way that can be related also to a physical machine. The products of this modeling can be thought of in the context of automata; to avoid ambiguity with automata of well-known formalizations, we offer the term abstract machines. Thus we need to familiarize ourselves with the notion of abstract machines as a "language" in the broad sense of a means of communication, in the same light as the verbal structures used in digital computers and the hardware structures used in analog computers.

We are familiar with phrase-structured languages and with mathematical languages because of education. If we were never taught about them, we could hardly spontaneously develop something similar in a single individual life.

We do not think of abstract machines as a language because we are not taught in that sense. But from the discussions in chapter 2 it appears that it is the one for which we have a natural inclination. It results from our sensorial experience since birth, and from the continuous realization of the cause-effect relation. To think of imaginary (abstract) objects in reciprocal relations is a natural activity. Children's drawings are an example. The sketches we make on a piece of paper, or on the blackboard, to help describe a difficult problem are another example.

A language is formed because of practical necessities or conveniences; and from these, the characteristics of the language derive. The spoken languages developed for communicating all kinds of information, in a noisy environment, without need of actual rigor. These languages are very flexible, all present a remarkably constant amount of redundancy, and all have some degree of ambiguity.

The mathematical languages came out for the complementary need of communicating with rigor a well-delimited type of information, in a protected environment. These languages have very little flexibility, tend to eliminate redundancy, and do not admit ambiguity.

For the purpose of communicating with computers we may imply a quiet environment, we need rigor, we cannot accept ambiguity, and we strongly desire a flexibility that can follow the multifacets of human thinking.

Programming languages of digital computers are sharing more and more the potentially infinite power of verbal languages. However, we have to recognize that for practical necessities they do not excel in flexibility. Moreover, they imply the complex translations discussed in chapter 1. The procedures used in analog computers are very effective, but they cannot compete with the generality and elegance of the phrased languages. In our approach, we bring the physical computer to take the same structures of a symbolic system that has the role of a language. The interesting results obtained by applying this approach have motivated this study.

In order to utilize the symbolic system of chapter 3, of which we implied a corresponding physical system in chapter 4, we need now to define a detailed set of symbols, rules, and conventions for documenting the productions in that system, that is, the abstract machines. In other words, we need to provide the means for the external representation of the abstract

machines devised by the user. Note that the means for the external representation do not constitute, and do not constrain the programming language, which is the substratum where the user develops the abstract machines.

We do not know whether it can be practical to bring the physical substratum of chapter 4 to the precise level that is appropriate for the user; or, put the other way around, whether it is practical for a human user to conceive abstract machines at the level of economical physical substrata. Therefore, we keep two successive levels, the same that were indicated as points B and C in Fig. 1 of chapter 1. For these two levels we introduce:

(1) A user language - A format, a set of rules, symbols, and conventions for representing the abstract machines, so as to form "a working language" and "a guide for all hardware representations".

(2) A machine language - A set of symbols and rules for representing the abstract machines in a form that can be accepted and understood by actual machines (the physical substrata), that is, hardware representations.

The two levels do not represent different languages. They are external forms of the same language that are oriented to different users, one human, and the other, electronic. Because of this different orientation, they may differ in some characteristics. What is important is that one form can be directly derived from the other, because of the isomorphism between the two substrata. In the examples of chapter 6, different degrees of proximity between the two forms can be observed.

In the ALGOL 60 Report (Naur 1960) different levels of language were recognized; namely, a Reference Language, a Publication Language, and Hardware Representations. Our levels (1) and (2) have roles similar to those of the Reference Language and Hardware Representations of ALGOL. To outline these similarities, we put quotation marks at the terms that have the same application as in the ALGOL 60 report. However, there are some differences. For instance, in ALGOL a hardware representation is obtained by translating a phrase structure into another phrase structure; here, it is obtained by transforming a multidimensional structure into a string of symbols, as discussed in section 3.4.4. However, the one-to-one correspondence between the elements of the two representations always holds. In the ALGOL

Reference Language, the entire structure of the program is expressed exclusively by means of lexical and syntactical characteristics; here a great recourse to spatial relations and to graphic means is made. In ALGOL, both the Reference Language and Hardware Representations provide source programs; the object programs are a quite different affair. Here, the hardware representations provide, in general, actual object programs.

To take account of all these facts, we call level (1) the "user language" and level (2) the "machine language". Their precise relation to high-level, reference, implementation, and machine languages of conventional programming will be discussed in section 5.4.1.

5.1.2 - Preliminaries on the user language

There is no doubt that the form of expression is very important. It is well known to psychologists that languages further in the child the development of some classes of mental structures and not others. In pedagogy, the cases are well known in which traumatic experiences with high school mathematics have erected barriers around that part of the cognitive field labeled "abstract symbolism" (Inhelder and Piaget, 1959, p. vii). Pager (1973) points out the inconveniences of information represented exclusively in phrase form, and suggests augmenting the language by means of various devices.

In programming the influence of language is undoubtedly no less determinant. Solutions are conceived or not, depending on whether the programming language allows their construction and representation or not. A page of listing of today's programs would not particularly attract a nonspecialist to develop a symbiosis with the computer. Echoing Whorf's (1956) hypothesis that the structure of the language influences the manner in which humans understand reality and behave, one can simply mention the possible influence on those who use computers of changing from a command language to that of mental images. At least, when programming in the form of abstract machines, the user would not feel himself to be a slave of the computer since he would have designed its characteristics.

Fortunately, the phrase language is merely one particular instance of the semiotic or symbolic function (Piaget, 1971, p. 46). The discussion

on modeling and representation made in section 2.2 now comes to fruition. We noted there, in several different contexts, the recurrent use of images and words in the common task of transmitting complex information, that is, the simultaneous use of different forms of expressions. Obviously, it would be advantageous for the programming languages, which have to represent so many varieties of processes ranging from mathematical computations, to payroll preparation, to modeling of intelligence, to take advantage of a plurality of forms of expression.

Forms of representation and of modeling are not independent. Each one, in a sense, preselects the other. Here there are two facts that favor a large latitude of representations. In the first place, the modeling is in the context of automata. As noted in section 2.3, automata have appropriate representations in a variety of forms, such as algebraic systems, verbal structures, and operational constructs. In the second place, here we use a physical substratum isomorphic to the symbolic substratum. It is precisely the use of these two isomorphic substrata that frees us from the severe constraints posed by the automatic translation of the source programs into the object programs - constraints that dictate the exclusive use of a formal, phrased language.

It is highly desirable that the form of representation can keep as much as possible the flavor of the abstract machines. From all the discussions in chapter 2, we can readily assume a basic representation for the abstract machines in the form of a kind of state diagram. Diagrams are not new. In automata theory, the state diagrams are a well-recognized form of representation. In programming, flow charts are highly recommended (but scarcely used). Schemas are a new form of analysis in computation theory (Manna, 1973).

One observation is in order here. Graphic means have been used so far as an accessory, auxiliary representation of certain aspects of a process, typically the control structure. Here, instead, the diagram will be the reference representation. Other forms will be derived from it when necessary. We will, also, always keep present the complementarity and the collaboration of the two main symbolic systems in the mental processes. Therefore, we will formulate a global representation of all the aspects of a process, by using simultaneously graphic and verbal structures, taking advantage of the first

especially for representing the dynamics and the constructs of the process, and of the second especially for representing single characterizations.

The fact that the reference language uses graphic means as a main constituent does not appear to imply impracticality, although present programming languages avoid graphic means. As discussed in section 3.4.4, our state diagrams can readily be expressed in the form of symbol strings, when necessary for machine communication. If a need emerges for automatic equipment that can accept and manipulate graphic expressions, undoubtedly industry will provide it.

Processes modeled in the form of abstract machines have a natural representation in graphic form. If, moreover, graphic means favor the imagination of the user, there is no reason for not using them.

5.1.3 - An experiment in applying modes of thinking to computer feature

In chapter 3 the basic structure of a symbolic substratum was given; however, detailed features of the substratum were omitted. This was done purposely, because it is important that these detailed features are carefully chosen in accordance with psychological characteristics of the users, rather than being formulated arbitrarily; and this needs long experience. In this chapter, a minimum set of necessary details are described, as they are at present being worked on. One of these features has been developed in terms of psychological characteristics, as reported in this section.

One of the basic elements of our substratum is the transition function. The effectiveness of a wealth of transition functions for describing complex processes in a simple form was recognized since the first use of the CPL machine. But transitions can be easily recognized also in our mental processes; transitions between items, times, actions, situations, places, states of mind, etc., are familiar and natural for everyone. Even without a specific psychological background on the subject, we felt that an investigation on the modes in which transitions occur in thinking could be useful for providing appropriate transition functions to the symbolic substratum.

A variety of people with different activity, age, and training have been interviewed. The form here shown was used to facilitate the extraction of the wanted information from the interviewed people. In order to bring the interviewee to the issue, transition is presented first as a physi-

cal transfer from one place to another. Eight modes (upper part of the form) are presented as applicable to planning a trip, for business or pleasure. The relative applicability of these modes is asked and results recorded as ordered numbers in the form. At this time the person being interviewed is already "in" the notion of transition, and sometimes is able to mention some other ways of looking at it, ways that are transcribed if different from the eight examples given.

Then the notion of transition is presented in the sense of "changing mind", "changing situation", "changing status". The various reactions, comments and sayings of the interviewee are interpreted and if an interesting pattern of the transition appears it is noted in the second half of the form.

The population interviewed included students, professionals in different fields, and people in a variety of occupations; their ages ranged from 11 years to mature age. The following is a sample of some relevant expressions obtained.

1. I will stay there for a certain number of days.
2. I will go there, and then I will see.
3. I will go to C, and I might stop in B.
4. I will stop there for a certain time.
5. Temporary block.
6. Several plans performed sequentially.
7. Canceling the plan, and making another.
8. Discuss this, before you forget.
9. If I think many things at a time, the efficiency decreases.

Of great interest have been the answers in regard to visual or verbal thinking, and serial or parallel mental activity. Undoubtedly, a wide variety of forms of thinking exist, and at the same time many common modes are used. Comparable numbers of people said that they think in images or in words; but after further introspection, said that actually they were thinking in both terms. Few people felt that they were thinking definitely either in images or in words. The majority of people reported that their cogitation is serial. A very few, however, claimed that their mental attention is completely parallel.

Expressions such as 1 in the list has suggested the STAY function (see section 5.2.4). Expressions such as 4 suggested generalizing the stopover transitions to include temporary staying. The emphasis expressed by people on the conditionality and priorities in stopping and changing plans produced the development of features for specifying priorities and options on several conditions.

The work done in connection with the reported interview has further confirmed the effectiveness of the transition functions in modeling processes, has contributed to the formulation of these functions, and has demonstrated, at least for this case, the possibility and the convenience of modeling computer features after common features of people's thinking.

	age	sex	profes	
	date	place		
	simple transition			
	condit. "			
	one stopover			
	several stopovers			
	conditional stopover			
	a plan			
	several tentative plans			
	sudden diversion			
	specification of each step	natur.	freq.	
	automatic sequence of "			
	change of mind			

5.2 THE USER LANGUAGE

5.2.1 - Structure of a program and its representation

For the CPL system, programing is to devise an abstract machine that performs a desired process.

These abstract machines are derived from the mental image that the user forms of the processes. These images should be channeled, oriented to the symbolic substratum of chapter 3, to yield the development of appropriate abstract machines in the substratum.

These machines are composed of elements from given collections, and are assembled in terms of certain mechanizations. These collections and mechanizations constitute the user language.

A program consists of the description of an abstract machine. These descriptions are made up of symbols, graphic and verbal, that indicate the elements used and the mechanizations involved. Each of these elements and mechanizations are simple in themselves. Complex processes can result from particular structuring of the abstract machines; however, the description of the machines remains at the level of the simple elements. Here we do not describe the execution of a process, but rather we describe a machine that will perform the desired process as a consequence of the particular structure we give the machine.

The symbolic (as well as the physical) substratum allows two types of structures: operational structures, the FSMs; and data structures, fundamentally the pages. A process is the outcome of the interplay of these two types of structures.

The user is given a standard frame for developing these interplays: the spatiotemporal frame represented in Fig. 13. The FSMs and the data structures merge in the programmable network PN. An FSM is a set of states that are operational structures and commands upon data structures. Primary data structures are sets of pages, each with a key which refers to a state of an FSM. When a page comes into PN, it implements a portion of an FSM, a state. In turn, that portion of FSM acts on the page, and optionally affects also the data structures.

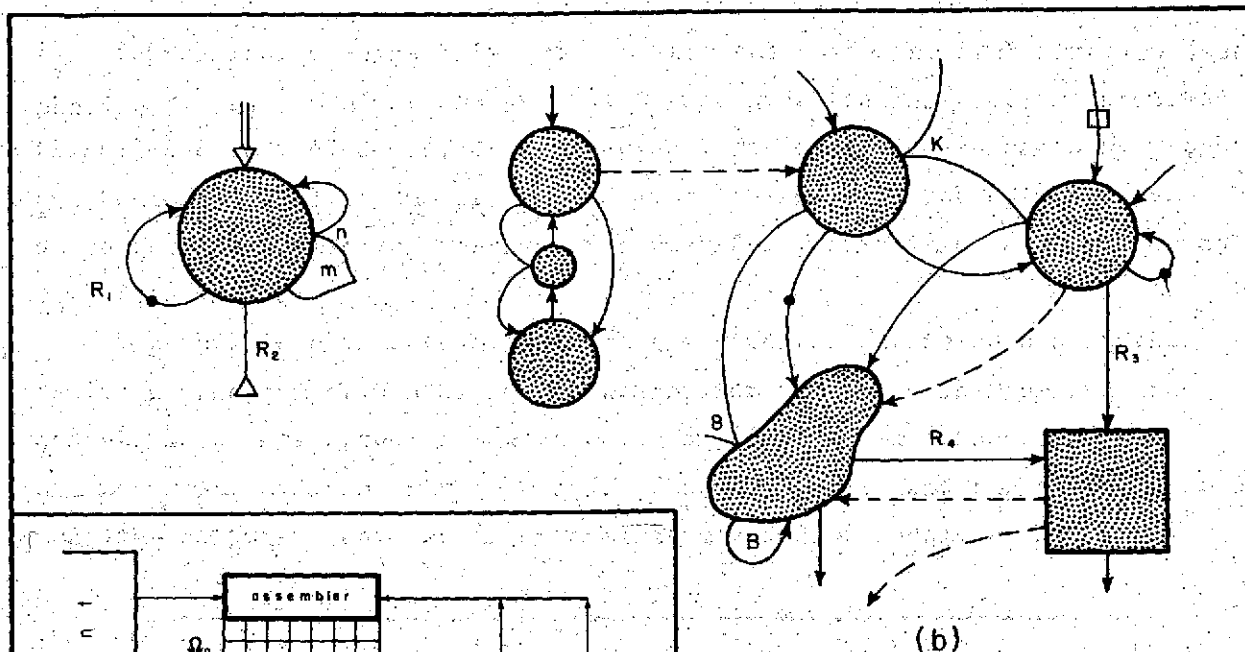


Fig. 17 - FSM structures

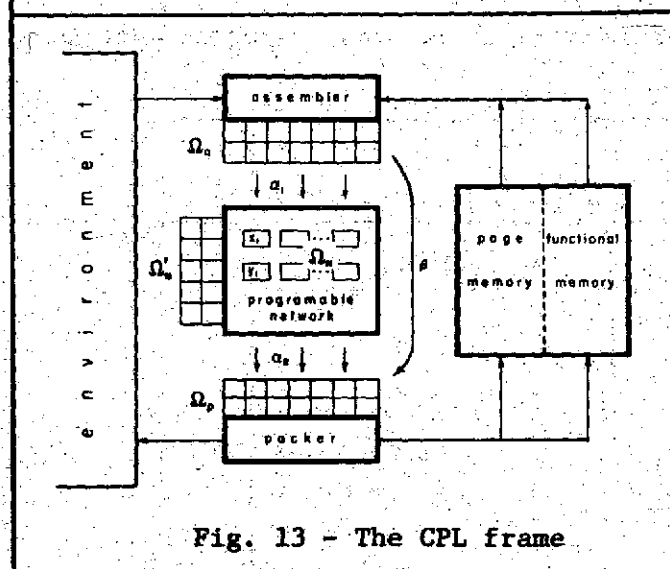
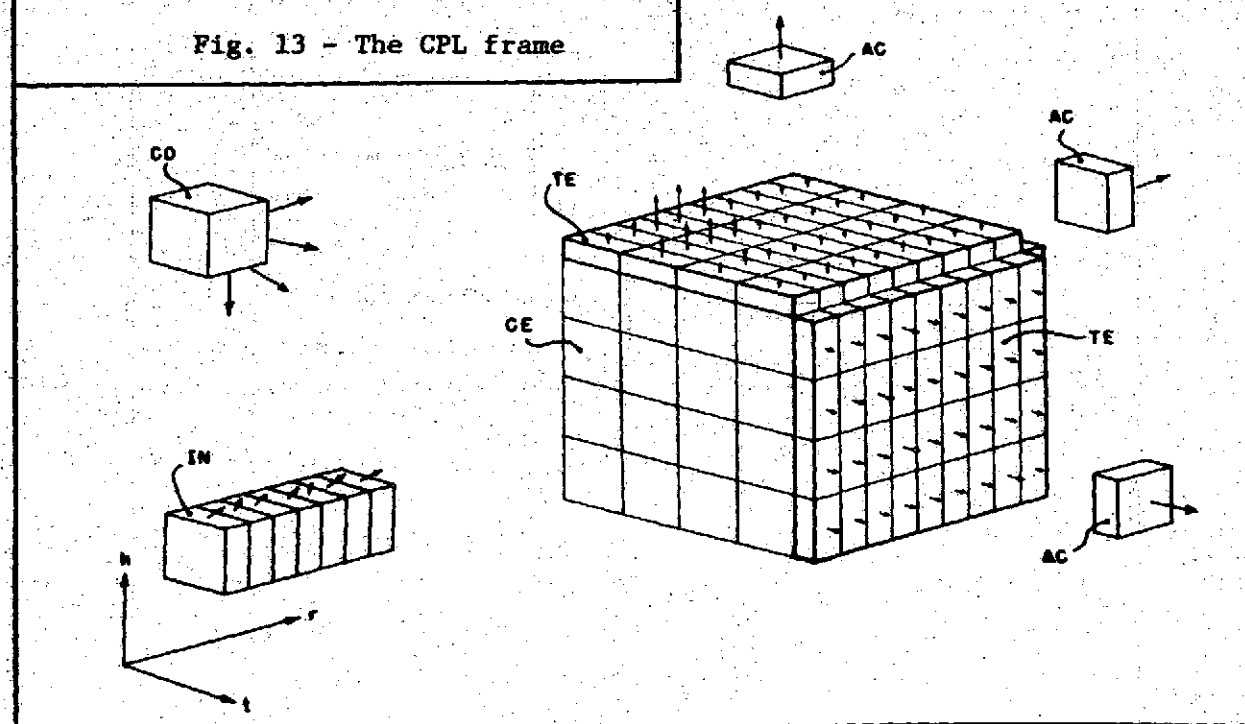


Fig. 18 - Page structures



Simultaneously, a flow of data from and to the environment is available at the assembler and packer points.

An auxiliary page storage Ω'_N is available in PN for facilitating the data transformations and for interactions among the pages.

An auxiliary functional storage medium is available in the memory for special data structures that can interact with the pages.

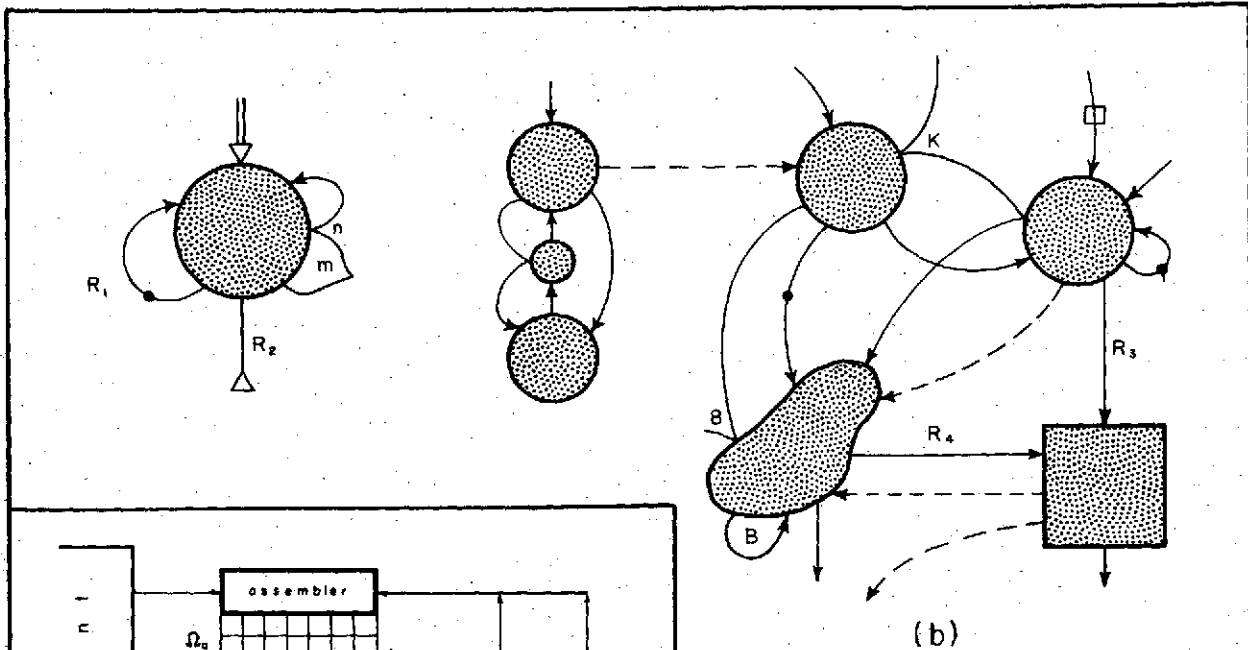
An FSM is represented in the form of a state diagram, with graphic and alphanumerical means (Fig. 17). The states are indicated as domains encircled by a line (dashed areas in the figure), within which the components specific to the state are written. The paths of the transitions are indicated by arrows and special symbols in the space separating the states. Components related to transitions are also indicated in this space.

An FSM can consist of one isolated state, a group of connected states, or even several groups of connected states. Whether to consider separated groups of states as independent FSMs or as one large FSM is matter of strategy in modeling a process.

A state is composed of the four components I, F, T, and R, some of which may be not present, and some of which may be composed of several sub-components. The modes and rules for describing these components are indicated in the following sections. Examples of state diagrams are in chapter 6.

The pages are created, transformed, rearranged, and eliminated by the FSMs, by actions of the I, F, T, and R. The FSMs can act insofar as there are pages that implement them. For the start of each program, obviously, an outside intervention has to inject at least a first page into the frame of Fig. 13. It should be noted that in this frame the pages are to call for pieces of program (the present state), rather than programs to call for data.

The pages are organized in different ways to conform to the data structure of each process. These organizations are implicitly described by the FSMs. However, they can be independently represented, if that is appropriate for visualizing or documenting the program. Pages can constitute a plurality of independent jobs; a linear array of similar sets of data (as in the program of section 6.2.1); a two-dimensional array (as in the program of section 6.4.1); or a mixture of various structures, as in the representation of Fig. 18 (which refers to a program discussed in section 6.4.2).



5.2.2 - INPUT PRESCRIPTION I

- Three categories of input objects are considered:
 - quantities* that are written in the program itself;
 - quantities* that are in some storages of the machine, which are referred to with names such as M, N, P . . .
 - quantities* that come from outside sources, which are referred to with names such as S_1, S_2, S_3

- k new inputs u_i are possible for any given CPL machine, say

$$u_1, u_2, u_3 \dots u_k.$$

- An input prescription consists of an ordered list of assignments for the u_i

example:

$S_4 - - 8 M 0$

it means: u_1 will have the present value** of S_4

u_2 will be empty (= 0)

u_3 " " " "

u_4 will be = 8

u_5 will have the present value** of M

u_6 will be = 0

the following u_i will be empty (= 0)

In the state diagram, usually, the input prescriptions are not indicated, because the chosen input quantities are apparent in the descriptions of F and T. But they should be kept in mind in regard to which set of inputs is available in each state. The input prescription is compulsory in the machine programs.

* The physical substratum is supposed digital, at least for the user's viewpoint. Thus all quantities are treated as numbers. Of course their meaning varies from process to process, and even from moment to moment in the same process.

**Present value, in this context, means the value that the source possesses when the page in question is activated, namely, when the page enters the assembler.

5.2.3 DATA TRANSFORMATION F

The variables and data available to the programmable network are:

- (1) x_r process variables in the page (in Ω_N)
- (2) x'_r variables in the auxiliary page array (in Ω'_N)
- (3) u_r new input data
- (4) x''_r variables in the following page (in Ω_a)
- (5) x'''_r variables in the previous page (in Ω_p)

(1) and (2) are the variables that can be transformed; (3), (4) and (5) can only be read. In the programmable network, variables (1) and (2) assume a new value as a function of themselves and of other variables and data in the network, as indicated in the general expression (3.2), here repeated:

$$X(i+1) = F [X(i) , U(i)] .$$

The usual way of expressing functions often involves a sequential application of simpler operations. Correspondingly, a sequence of different network configurations is used, within the cycle of a page in PN, to perform the total function F.

Here, the mapping (3.5) of page 80 has to automatically provide these configurations from the user's expressions. A form is suggested for these expressions that, on the one hand, matches a mode of thinking, and, on the other hand, matches the characteristics or constraints of a physical network. First, the variables that are to be transformed are named; second, the type of operation to be performed is indicated; and third, the variables and data, if any, that are read for obtaining the transformation are named. It is understood that the reading of the variables is made before that the transformation takes place; if the same variable appears in the first and in the third parts of an expression, its old value is implied in the third part, while the first part represents its new value. This corresponds to the assumed convention of time, as indicated in Fig. 9 at page 74.

The variables will be indicated with specific names starting with a capital, or with capital letters (A, B, C, ...). The variables in the auxiliary page array, in the following page, and in the previous page will be indicated in the same way but with the addition of one, two, and three primes, respectively. The new input data will be indicated with specific names, all in lower case, or with lower case letters (a, b, c, ...). Because of the syntax of the language, the form chosen for the expressions, and the use of specific allocations in the state diagram, the lexical characteristics are here not critical as in conventional programming languages.

The functions are indicated with symbols, as suggested in the following. Because the user language is the state diagram, it is important that the functions be indicated in a concise, possibly selfexplanatory form. It is essential that the meaning is given without ambiguity; but the lexical characteristics are irrelevant; the symbol rigidity required by a computer is postponed to the time of the actual coding of the program. The functions that can be used depend on the implementation of the language. The approach taken here is to allow the user to develop a program as much as possible in his own terms; then, gradually, the program is refined in terms of a specific implementation of the language. Context dependency is here highly beneficial.

The CPL system has the facility of producing data transformations globally, by means of operational networks, rather than by means of sequences of commands. Therefore, the language should give the user the control of this facility. The approach is again that of using a correspondance between possible structures of thinking and possible structures of the physical substratum constituting the programmable network.

When we think of a network, we look at it as a parallel array, if there is a regularity of repeated characteristics; we look at it as composed of a main element and collateral parts, if all operations relate to a single variable; we look at it serially, one part at a time, if the network is composed of a distributed set of elements performing different, dependent, or independent functions; and we think of it as a unit, without entering into details, if we are already familiar with what it does. According to these natural ways of thinking, and in agreement with the structure of PN, four modes of prescribing data transformations are established.

Mode 1 - Collective prescription (an assignment of a similar function to several variables x_r).

In the user language, this prescription is made by listing the names of the variables involved, followed by the symbol of the operation, and by the arguments in an order corresponding to that of the variables.

Example:

$$A B E F \sum a b c d$$

This mode correspond, in a sense, to the vector operations, but it is more flexible. As an example, in the expression

$$A B C D A' E \sum a b c D' A F$$

ABC and abc can be viewed as components of the vectors X (the main variables) and U (new input data), respectively; but A' and D' are components of the same vector X' and appear with a different role in the expression; moreover, E (component of vector X) accumulates F which is another component of the same vector X. Such an eterogeneous composition of variables does not pose problems to the machine, because of the isomorphism between structure of the language and structure of the physical substratum.

Mode 2 - Multiple prescription (different functions assigned to different variables).

The description of each function for each variable is made as in Mode 1, and an indication is added signifying that the functions are executed as a single network. In the preferred user language, this indication is a square parenthesis enclosing the entire prescription. As an example:

$$[A \sum a, B = b, C \times 2, D + 1]$$

Obviously, in this case, attention should be given to the relation among the variables; these questions appear clear themselves if the user thinks of a physical network. For instance, the following prescriptions are perfectly valid

$$[A \sum B, B = A]$$

$$[A \times 4, B \sum B', A'B' = A B]$$

In the first prescription, B will hold the previous value of A, and A the sum of the two. In the second prescription, A' and B' will show the original values of A and B, respectively, while A will double and B will sum the original value in B'.

Mode 3 - Individual prescription (a complex function centered to a variable).

If only one variable is to be transformed, the entire network can be mobilized for a particular operation, without the possibility for ambiguities. Simple examples are:

$$A \Sigma \log B$$

$$(CD) \Sigma C$$

In the second example, C and D are treated as a single variable that accumulates the most significant part of itself; configurations of this type can be advantageously used for finite approximations of differential equations.

Mode 4 - Special transformations (that were developed for specific recurrent tasks).

In the use of a computer often a group of operations are used and repeated in many recurrent jobs. It is thus desirable, for the simplicity of the programs, or for the speed of the execution, to have that group of functions executed by means of a single specialized network. In these cases, the complex network is not described any longer, but simply called by means of a coded word.

In conventional programming, a specific data transformation has to be built in terms of the functions available in the programming language, supplied with auxiliary commands. Only in particular cases, the desired data transformation can be framed as an executable statement; in most cases, a specific sequence of statements and control commands -- a routine -- has to be devised. A similar procedure could be used also here; however, the presence of an entire page in the network and its programability make a more direct implementation possible. A larger number of desired data transformations will correspond directly to expressions that can be formed in one of the four modes. A succession of such expressions in a state gives the state a significant processing capability. In this context, a state will in general correspond to a phase of the process as viewed by the user, as testified by many of the examples in chapter 6.

The establishment of specific functions to be made available in each mode requires a thorough study of algorithms and programmable networks. General

outlines and specific implementations are given in the following.

Mode 4 represents the hardware implementation of recurrent special data transformations. The interesting point is that, when a programable substratum is available, these implementations do not require hardware interventions; they can be implemented by the user, when needed, with a preliminary construction in terms of the other modes.

Mode 3 represents functions that are not a uniform attribute of the substratum; that is, functions that either cannot be performed simultaneously for all the variables, or that are available only to some privileged variables. This mode is the one most closely related to conventional microprogramming. Functions in look-up-table form are to be prescribed in this mode also, because only one table per function will be generally available. In the CPL 1 machine, this mode has been implemented for some hard-wired functions at privileged variables.

Modes 1 and 2 refer to uniformly programable functionality of the substratum; that is, functions that can be implemented simultaneously and independently for all variables. The distinction between mode 1 and mode 2 is introduced only for simplifying the language that describes the networks. Mode 1 has been analyzed and implemented to a certain extent; in the following, samples of the richness of functionality achievable in a programable substratum are shown; working notations that are used in the state diagrams are also indicated, in the form of examples.

Setting

- To zero (clear). Notation: $A \circ ABC \circ$
- To the maximum value (binary 111 ...). Notation: $A \max$
- To the absolute value ($A + |A|$). Notation: $A ||$
- Complement (Boolean function). Notation: $A \text{ compl}$

Shifting

- To the right for n binary positions. Notation: $A : k$ ($k = 2^n$)
 - To the left for n binary positions. Notation: $A \times k$ ($k = 2^n$)
- Options: (1) Several words cascaded; notation $(AB) : k$. (2) Injecting one rather than zero; notation $A : k$. (3) circular; notation $A \odot k$.
- Bit reversal. Notation: $A \text{ rev } B$. Interpretation: A assumes the reversed patterns of bits present in B, and B remains unchanged.

Increment

- The variable is increased by one. Notation: $A \ B \ C \ +1$. Interpretation: $A \leftarrow A + 1$, $B \leftarrow B + 1$, $C \leftarrow C + 1$; when a variable reaches the maximum value, no further change occurs even if prescribed.
Option: the variable overflows and restarts from zero; notation: $A \oplus 1$
- The variable is decremented by one. Notation: $A \ B \ C \ -1$. Interpretation: $A \leftarrow A - 1$, $B \leftarrow B - 1$, $C \leftarrow C - 1$; when a variable reaches zero, no further change occurs.
Options: (1) the variable assumes negative values and stops at the maximum negative value; (2) when the variable reaches zero, it starts again from the maximum value.

Transfer

- Copy of a value. Notation: $A \ B \ C = D \ E \ F$. Interpretation: $A \leftarrow D$, $B \leftarrow E$, $C \leftarrow F$; D , E , and F unchanged.
- Movement. Notation: $A \ B \ C \leftarrow D \ E \ F$. Interpretation: as above, but D , E , and F are cleared.
- Interchange. Notation: $A \ B \ C \ \leftrightarrow D \ E \ F$. Interpretation: $A \leftarrow D$, $B \leftarrow E$, $C \leftarrow F$, $D \leftarrow A$, $E \leftarrow B$, $F \leftarrow C$.

Selection

- The largest value. Notation: $A \ \text{larg} \ AB$. Interpretation: A assumes the largest of the values in A and B ; B remains unchanged.
- The smallest value. Notation: $A \ \text{smal} \ AB$. Interpretation: corresponding to that above.
- Mean value. Notation: $A \ \text{mean} \ AB$. Interpretation: $A \leftarrow (A + B)/2$, B remains unchanged.

Accumulation

- Summation. Notation: $A \ B \ C \ \Sigma \ a \ b \ c$. Interpretation: $A \leftarrow A + a$, $B \leftarrow B + b$, $C \leftarrow C + c$; the variables do not exceed the maximum value, positive or negative.
Option (1): cascaded variables; notation: $(AB) \ \Sigma \ D$; interpretation: D is accumulated in A , B constitutes a continuation of A .
Option (2): free overflow; notation: $A \ B \ \uparrow \ a \ b$; interpretation: when each variable reaches the maximum value, the accumulation continues from zero value.
Option (3): the variables do not assume negative values, and stop at zero.
- Subtraction. Notation: $A \ B \ C \ \Sigma - \ a \ b \ c$. Interpretation: $A \leftarrow A - a$, $B \leftarrow B - b$, $C \leftarrow C - c$; the variables do not exceed the maximum values.
Options (1), (2), and (3) as above.
- Absolute value of the difference. Notation: $A \ |\ - | \ B$. Interpretation: $A \leftarrow A - B$, if $A > B$; $A \leftarrow B - A$, if $B > A$; B unchanged.

Product

- Limited precision. Notation: $A \ B \ C \ \Pi \ a \ b \ c$. Interpretation: $A \leftarrow A \times a$, $B \leftarrow B \times b$, $C \leftarrow C \times c$.
Options of different precisions in privileged variables.

Boolean functions

- AND. Notation: A AND B. Interpretation: each bit of A assumes the value of the AND function of the corresponding original bits in A and B; B remains unchanged.
- OR. Notation: A OR B. Interpretation: as above, but using the OR function.
- Exclusive OR. Notation: A EOR B. Interpretation: as above, but using the exclusive OR function.

One of the basic characteristics of a programmable substratum is the connectivity. The connectivity poses constraints to the expressions above described. In reference to Fig. 14c, we can define a distance d between two variables x_i and x_j by counting the number of variables from x_i and x_j (included), in a given order. Therefore, for any given physical substratum, a constraint will be in the language, for which d in the expressions of mode 1 should be not greater than a given value k . The value k may be different in the two opposite directions in the sequence of the variables. Privileged variables will generally have larger k than the other variable; in Fig. 14d, the first variable does not have constraints of distance.

Function F in a state can be also devoted to activities to be performed in PN by other pages. This is one of the means for making concurrent work by part of several FSMs and pages possible. The driven transitions, section 5.2.4 (1)e, are prescribed as an F function by part of the FSM that orders the transition. The notations used in the state diagrams and their interpretation are as follows.

ST n	next page transfers to state n of its FSM.
ST n (k)	all pages of FSM k transfer to state n.
ST n all	all the following pages (of the present memory segment) transfer to state n of their respective FSM.

The driven transitions supercede any transition function T that may be described in the present state of the driven page.

Also routing prescriptions can be driven by a page other than the one that performs the routing. The driving page prescribes the routing as an F function. The notations used in the state diagrams and their interpretation are as follows.

RT k (. . .) The pages of FSM k implement the routing described.

The parenthesis stands for a routing prescription as indicated in section 5.2.5.

Function F in a state can include also commands. Of particular necessity are commands that control the output production. The following are given with their notation and interpretation, also with reference to the routings described in section 5.2.5 (4).

output (name)	: the variable "name" in the functional memory is transferred to the output buffer.
output SEG(k)	: the page segment k is transferred to the output buffer.
distribution	: the content of the distribution is transferred to the output buffer, and the distribution is cleared.
record	: the present content of the output buffer is recorded.
erase	: the present content of the output buffer is erased.
record open	: the present and future content of the output buffer is to be recorded.
record stop	: the present content of the output buffer is the last to be recorded.

5.2.4 TRANSITION FUNCTION T

A transition function is composed of two parts

- (1) - a set of paths through the states of the FSM
- (2) - a set of conditions (such as values reached by the variables, signals coming from the outside) that determine the choice of one path.

Moreover, a transition function can be

- (a) - described in the FSM itself
- (b) - be imposed by another FSM or be produced by an outside control (driven transitions).

There is a transition function in each state, and it is performed each time a page has a cycle in that state, after the data transformation F.

(1) PATHS IN TRANSITION FUNCTIONS

(a). No prescription of any sort for T function means that the FSM remains in the present state (until some action from the outside of that FSM occurs). In the state diagram, this case is represented by the absence of any arrow emerging from the circle representing that state, Fig. 19(a).

(b). Unconditional transition to the state with the next label (in the natural numerical order). Note that this is the simplest coded transition, no state labels need to be indicated. The adopted graphical representation consists of drawing adjacent the circles representing the states, Fig. 19(b).

(c). Unconditional transition to state h. This prescription needs simply the state label h. Its graphical representation consists of an arrow pointing to state h, Fig. 19(c).

(d). Transition to different states depending on conditions. The T function and the related state labels need to be prescribed. The graphical representation consists of several oriented arrows emerging from the state, Fig. 19(d). Details are given in (2).

(e). Driven transition (produced as a consequence of actions by part of some other FSM). No prescription is made in this FSM. The graphical representation consists of a dashed arrow, Fig. 19(e). Similar symbols are used also to indicate the starting state.

(f). Go to state h and stay there for n cycles (then the transition prescribed in state h will act). The prescription needs a code, the label h and the value n. The graphical representation consists of an arrow with open head where the value of n is written, Fig. 19(f).

(g). Stay in the present state for n cycles (then the other prescriptions will act). The prescription needs a code and the value n. The graphical representation consists of an arrow looping into the state with the value of n written inside, Fig. 19(g).

(h). Go to state h stopping over states k, m, . . . (stopover transition). The prescription consists of the multiplicity of state labels in an established order. The graphical representation consists of a jagged arrow with notches pointing to the states where stop is made, Fig. 19(h).

(i). As in h, but staying in the stopover states for assigned numbers of cycles. The prescription is as in h with added number of cycles. The graphical representation is as in h, with the number of cycles written in the notches, Fig. 19(i). Note that if the state where to stop is not specified, the FSM will remain idle for the specified number of cycles.

(j). Priority is assigned to one or more transition branches. Priority means that that branch (if chosen by the function T) will occur first, regardless of other conventions. The prescription consists of a code added to the description of that branch. The graphical representation consists of a dot superimposed to the arrow representing that branch, Fig. 19(j).

(k). Lock of page. This feature is used when the page movement is controlled by function T, rather than following the automatic circulation. The prescription consists of a code added to the description of the branches for which the page remains in operation. The graphical representation consists of a circle added to those branches, Fig. 19(k).

(l). End of page. This transition produces the disappearing of the page. It is described as a reserved state label. It is represented graphically by a triangle, Fig. 19(l).

The feature of the priority allows the mechanization of a variety of rules. In reference to stopover transitions, when the states connected with priority transition are adjacent (in the state-label order), the next destination of a stopover transition will be reached at the end of the priority transitions, as in example 3 of Fig. (j). When the states connected with priority transition are not adjacent, the occurrence of the branch with priority cancels every previous stopover prescription (ex. 4 of Fig. 19(j)).

Transitions f and g, and transition h in the second example given in Fig. 19(h) produce similar staying. But this redundancy is in fact a flexibility which eases the programing. In the case of transition g, the prescription of staying is made once and it holds for all the transits through that state. In the case of transitions f the stay can be different for different arrivals to that state. In the case of transition h, the stay can be different in accordance with events occurring in that state.

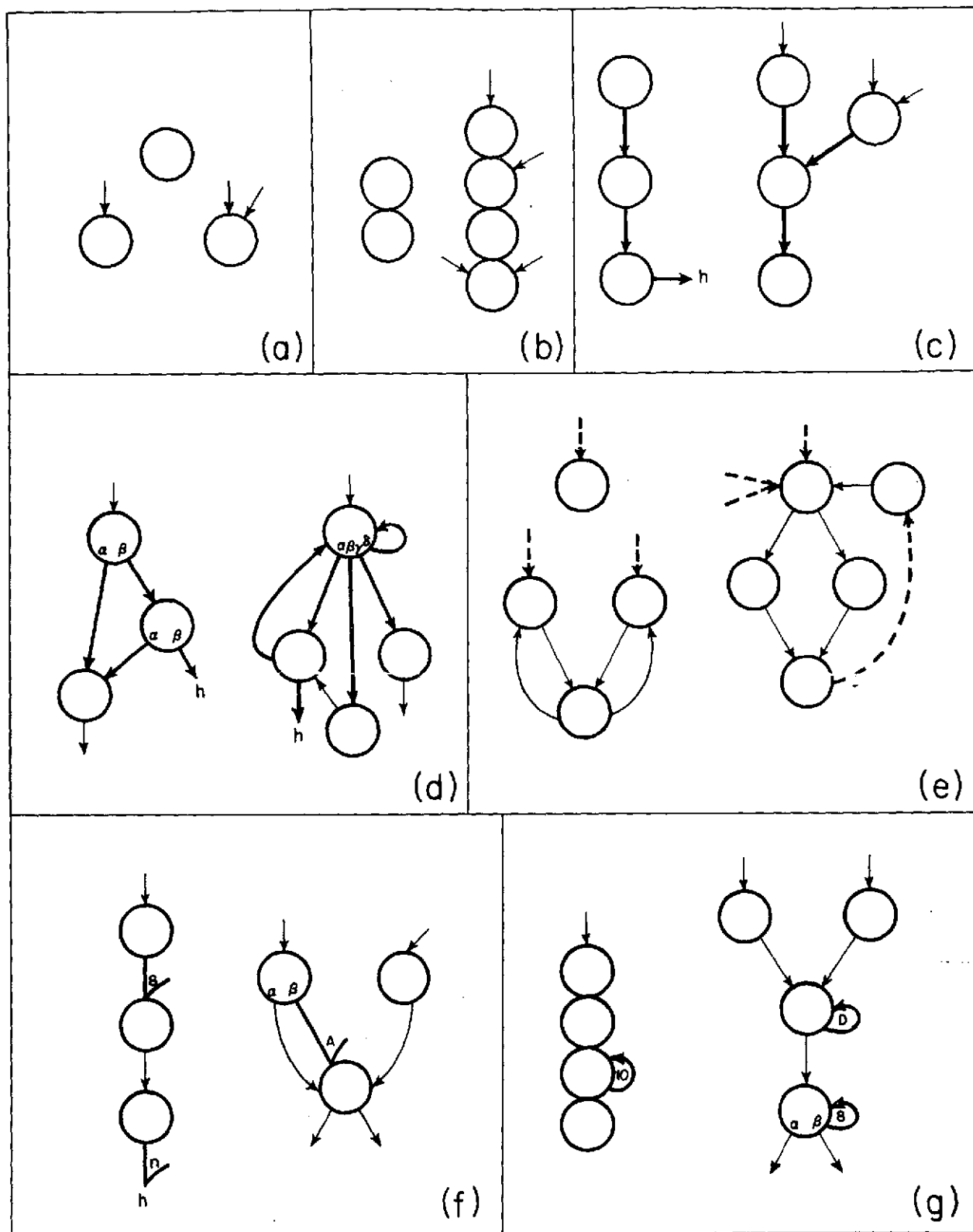


Fig. 19 ~ Symbols for the transition paths (cont.)

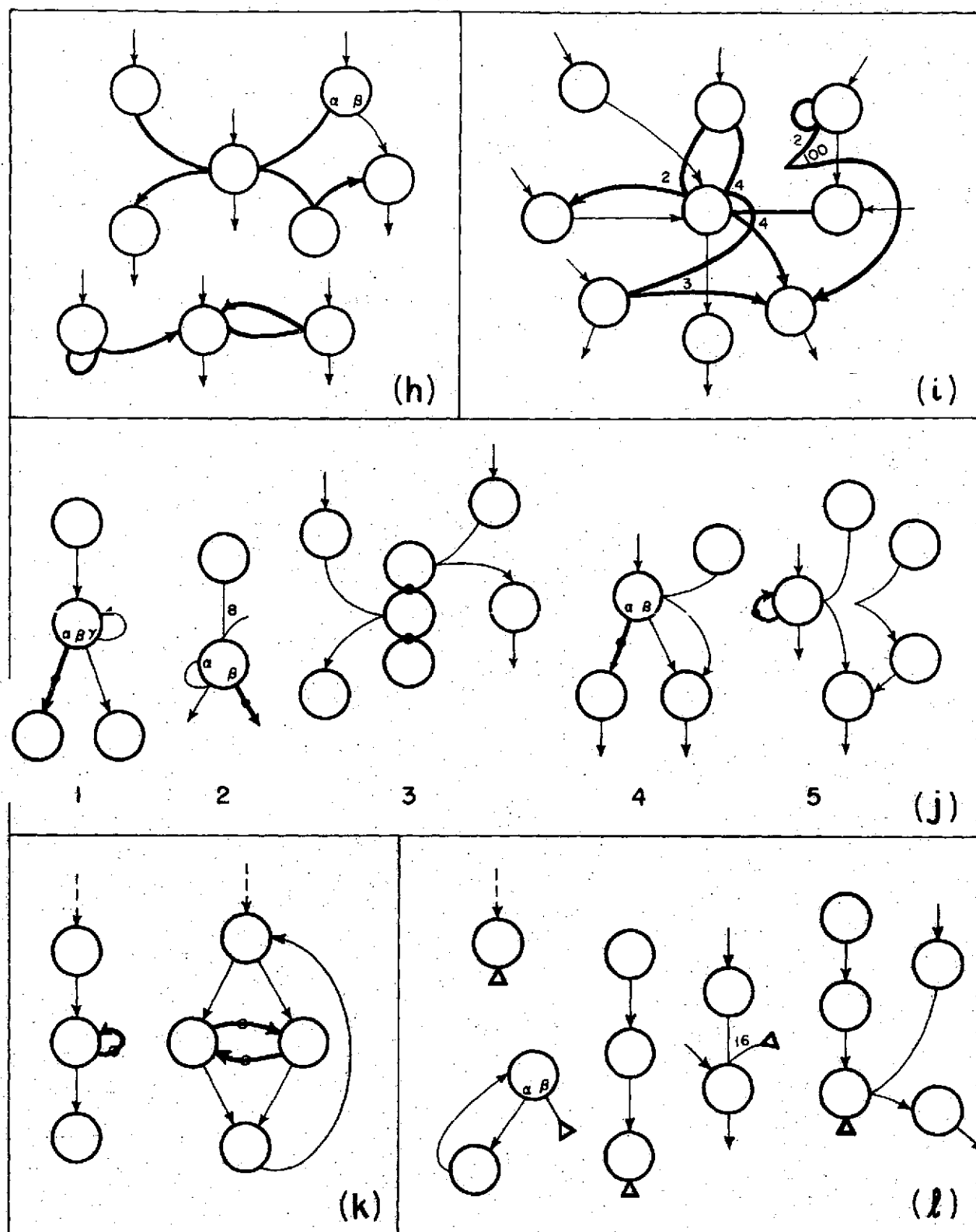
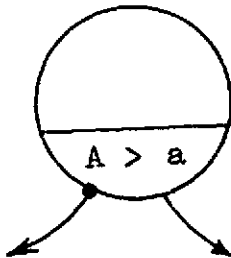


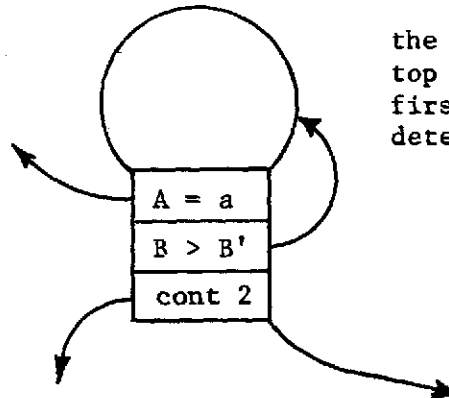
Fig. 19 - (cont.) Symbols for the transition paths

(2) CONDITIONS IN THE TRANSITION FUNCTION

The conditions that affect the choice of the path are expressed in conventional notations, below a horizontal line. Examples:

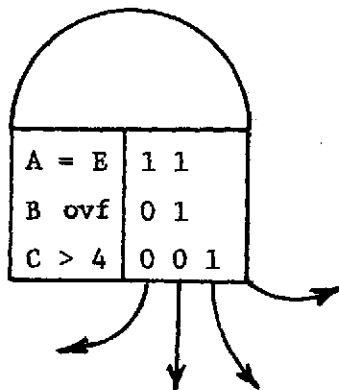


the dot indicates the branch taken when the condition is true



the tests occur from top to down, and the first that is valid determines the path

the branch from the corner is the path taken when no one condition is true (the "else" case)

Decision table form

no indication in the table means don't care

all tests are performed before a branch is chosen

The set of conditions is pertinent to each implementation of the language. The following set is adopted here: = , > , < , ovf (overflow) , and cont 1, cont 2, ... cont k (control signals appearing from outside, they can be indicated also by name).

5.2.5 ROUTING R

Variables x_i can be routed

- (1) to output (to an output buffer storage from which they will be recorded on the indicated peripheral at the prescribed time);
- (2) to different places in the page (where they will be found at the next circulation);
- (3) to other pages (where they will be found at the next circulation);
- (4) to functional memories (regions of the memory other than the circulating pages);
- (5) to control places (typically for structuring the memory).

Moreover, the routing can be

state dependent: it occurs every time a page is in that state - in the state diagram is indicated at the right of a vertical line;

transition dependent: it occurs only if a given path is chosen - in the state diagram is indicated beside the arrow representing that path;

driven routing: it occurs when another FSM produces that prescription - in the state diagram of the driven FSM is indicated in parentheses.


The routing operations occur after the completion of the work in the programmable network; (physically, they are performed when the page is in the packer).

The details of the routings are pertinent to the implementation of the language. In the following, the choices adopted here are described.

(1) ROUTING TO OUTPUT

- Simple indication of the name of a variable x_i means that the indicated variable disappears (becomes = 0) from the page, and its value is stored (in queue form, if not otherwise prescribed) into the output buffer.
- Indication of the name of a variable $\underline{x_i}$ underlined means that the variable is kept in the page, and its value is copied into the output buffer as above.

(2) ROUTING TO DIFFERENT PLACES IN THE PAGE

description (in the form of examples)	notation
<ul style="list-style-type: none"> - clear variables A, C, D - exchange A with D - shift all the variable to left, the last becoming = 0 - circulate all the variables of one position to the left 	<p>$A_0 C_0 D_0$</p> <p>$A \rightleftarrows D$</p> <p>\leftarrow</p> <p></p>

(3) ROUTING TO OTHER PAGES

- The name of a variable followed in parenthesis by the name of a new variable and of an FSM means that the routed variable will substitute for the new indicated variable in all the pages of the indicated FSM. Example: A(B,3).

(4) ROUTING TO FUNCTIONAL MEMORY

- Storage. Notations and interpretation as follows.

- A (name) : variable A in the page is stored as variable "name" in the functional memory; A in the page remains unchanged.
- A → (name) : as above, but A in the page is cleared.
- A (B) : variable A in the page is stored in the functional memory as a variable with name equal to the content of B in the page.
- A → (B) : as above, but A in the page is cleared.

- Acquisition. Notation and interpretation as follows.

- A = (name) : variable A in the page will acquire the content* of variable "name" in the functional memory.
- A ← (name) : as above, but the content of "name" will be cleared.
- A = (B) : variable A in the page will acquire* the content of the variable in the functional memory that has name equal to the content of B in the page.
- A ← (B) : as above, but the variable in the functional memory will be cleared.
- A read : variable A in the page will acquire* the content of the variable in the functional memory that has name equal to the present content of A.

- Exchange. Notations and interpretation as follows.

- A ↔ (name) : variable A in the page and "name" in the functional memory exchange their values.
- A ↔ (B) : variable A in the page and the variable in the functional memory with name equal to the content of B in the page exchange their values.

- Accumulation. Notation and interpretation as follows.

- A acc (name) : variable A in the page is accumulated to the present content of variable "name" in the functional memory.
- A acc (B) : variable A in the page is accumulated to the present content of the variable in the functional memory with name equal to the content in B of the page.

*that content will be found in the page at the next circulation.

- Distribution. Notation: $\{C$. Interpretation: variable C in the page is distributed in a distribution region defined in the functional memory. A distribution region is a sequence of n words reserved in the functional memory; these words are labeled $0, 1, 2, \dots, n-1$. When a variable with value k is distributed, the present value of the distribution word with label $= k$ is incremented by one; for any value k larger than $n-1$, the increment is made in the word labeled $n-1$.
- Count. Notation: δ (name). Interpretation: an increment of one is made in the variable "name" in the functional memory.
- Maximum. Notation: $D \max$ (name). Interpretation: the variable "name" in the functional memory assumes the largest of the values of D and of the old content in "name".

(5) ROUTING TO MEMORY CONTROL

- Creation of a page. Notation and interpretation:
 - $P \ n$ a page is introduced into the present memory segment, in state n of the present FSM.
 - $P \ (k)$ a page is introduced in the initial state of FSM k .
 - $P \ n(k)$ a page is introduced in state n of FSM k .
- Distribution definition. Notation: $\text{distrib } (n)$. Interpretation: a sequence of n words is reserved in the functional memory as distribution region.
- Change of page segment. Notation: $\text{SEG } (k)$. Interpretation: the circulation of pages transfers to the page segment k . A segment can be also a distribution sequence.
- Scanning. Notation: $\text{SCAN } (\text{code})$. Interpretation: the circulation of pages occurs along a segment, across an array of segments, across planes of segments, etc., according to the "code".

5.3 AN IMPLEMENTATION OF MACHINE LANGUAGE

5.3.1 - Outline

In this section, the machine language implemented in the CPL 1 processor is described. This implementation can be considered as a subset of the user language described in section 5.2. This subset reflects the particular contingencies and scope of the CPL 1 processor as described in section 4.4.

Because of the specific application for which the CPL 1 processor was constructed, particular attention was given to keep programs concise and simple. The modeling of the actual work of the machine in FSM form made this goal possible. For economy of peripherals, static card readers were adopted as an input device, and conventional punch cards were used as a program medium. However, much high exploitation of the information capacity of the 960 bits of the card was made, in comparison to conventional codings. Preassigned fields were established in the card with a one-to-one correspondence with the items of the FSMs, states, I, F, T, and R. In this way, the user can produce the machine programs directly, by looking at the state diagram (the user's program) and writing piece-by-piece on the card with a desk punching machine.

The feature of using no prescription for the most frequent choice was abundantly used. The binary codes were given a morphology similar to that of spoken languages, so that the user, after a short practice with the processor, could compose most of the codes himself without looking in the dictionary. The fixed format on the card made the syntax of the language obvious and no error prone. With the development of the system, however, the fixed format required an acrobatics for fitting into it larger amounts of information. Although this feature is not relevant to a general context, it has been of interest for exercising conciseness in the mapping (3.5).

In the particular context just described, the connection between user language and machine language discussed in section 5.1.1 did not pose problems. The user thinks of one language, that of the FSMs. The development of programs is carried out by sketching state diagrams. When a preliminary test is desired, portions of the state diagram are punched directly on a

card and fed into the CPL 1 machine. Observation of the results may suggest some modifications, which are usually written on the same card. During the development of a program, a card can be used as a small scratch pad. Items can be added and deleted (all ls), and new state descriptions can be written, until space is available in the card and there is a possibility of transferring to the new fields.

When a program is completed to the user's satisfaction, the updated state diagram constitutes the documentation of the program in the user language, and one or more cards constitute the original for the duplication of the machine programs.

What seems most significant in this experience, abstracting from the specific context and environment in which it occurred, is the man-machine interaction. When developing a program, we discuss the process in terms of states, transitions, and data structures; we talk to the machine in the same terms; and we can interpret the raw data of the computer in the same context.

5.3.2 - The format of the card

The words describing a configuration in the programable network constitute the most complex items in the FSM descriptions. Statistically, it was observed that in typical processes a configuration is used repeatedly in a single FSM. Therefore it was found advantageous not to write the words W_F in the state descriptions, but to write them only once in a separate field, labeled "common storage" (Fig. 20). In the state descriptions, words W_F are referred to by means of the number of the column in the common storage where they have been written.

The fields in the card are (Fig. 20): the common storage for fifteen 12-bit words W_F ; a column for the name of the FSM; a column for the code word; and sixteen fields, labeled from 0 to 15, for the description of sixteen states. The label of the state is given by the label of the field where it is written. Each state field is divided in four subfields: the top one for the input prescription I, the middle left for the calls of words W_F , the bottom one for the transition function T, and the middle right for the routing prescription R. State 15 is dedicated to the recording functions, and has a slightly different format.

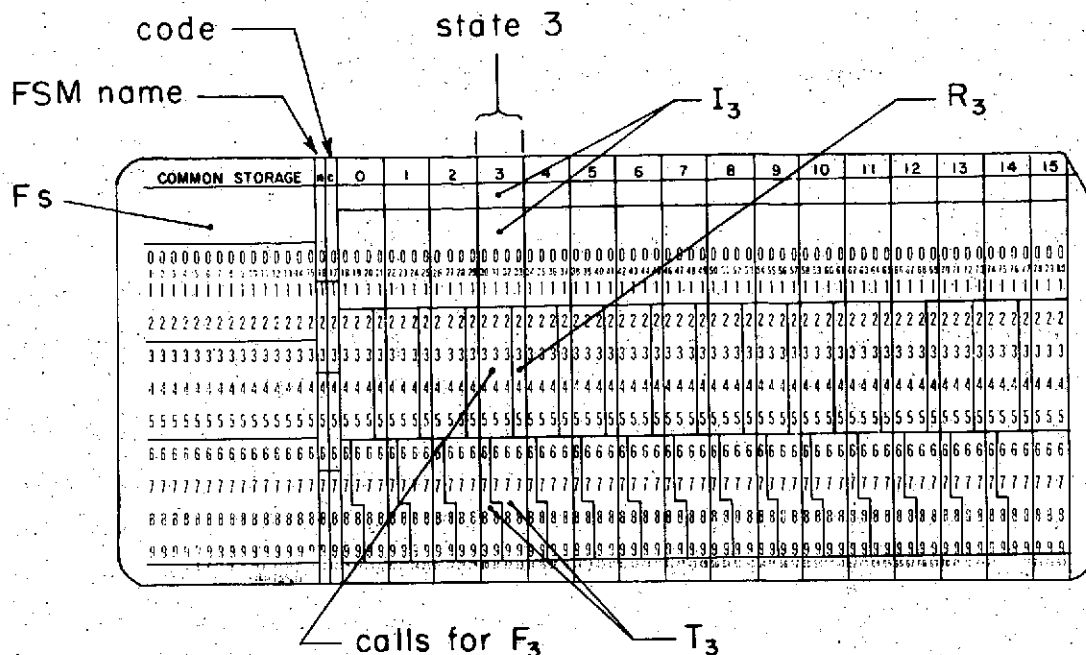


Fig. 20 - Fields in the program cards of the CPL 1 processor

Input prescription I. Each of the 4-bit words refers to the prescription of one u_r . The top bit indicates whether the input datum is written in the card or derived from an outside source. In the first case, the following bits indicate the mantissa of a number whose exponent is in a section of the code word. In the second case, the following bits select one of eight sources that are preselected among 64 by another section of the code word.

Data transformation F. The three 4-bit words in the call-for-F field indicate column numbers in the common storage. In the common storage, 12-bit words refer to configurations of the PN with the morphology described in section 5.3.3.

Transition function T. The first word prescribes the variables that are to be compared, in accordance to a coded table. The second word indicates the type of comparison, $=$, $>$, etc., and whether it is a stopover transition. The following two words indicate the states to which transfer in accordance with the result of the test. When a special code is present in the second

5.3.2

word, the first word indicates which outside control signal, or internal event such as overflow, is involved in the decision.

Routing prescription R. The small word for the routing prescription is augmented by a section of the code word. Routing to the output buffer, with or without clearing of the variables in the page, is indicated directly. Reserved codes indicate special routings, such as the transferring to a distribution function.

5.3.3 - The Morphology of the words

One of the most interesting issues of the CPL system is the mapping (3.5) between a symbolic description of data transformations and the codes given to the programable network for its execution. The peculiarity of this mapping is that it does not occur through the activity of a compiler that assembles sequences of instructions from a given instruction set as a consequence of the interpretation of sequences of statements from a formal phrase-structure language. The mapping occurs through operational networks that are devised by the user in a symbolic substratum. Because of the isomorphism between the symbolic and physical substrata, the execution can be obtained by a physical operational network that has the same objects and the same structure as the symbolic network devised by the user. In such a situation, the mapping (3.5) can be accomplished by means of a simple transliteration, if an appropriate language is provided to the user and to the decoders of the machine.

The CPL 1 processor, in spite of its elementarity, has been an interesting benchmark for experimenting with the essential points of this mapping. By using morphologies and inflections that are familiar because of their use in spoken languages, it has been possible to develop a direct correspondence between the bit patterns that control the programable network and the productions of a language that mean the desired data transformations to the user. By taking certain attentions, it is also possible to utilize almost the totality of the available bit patterns. It is possible to give the language such a structure that it appears to the user an effective means for communicating data transformations.

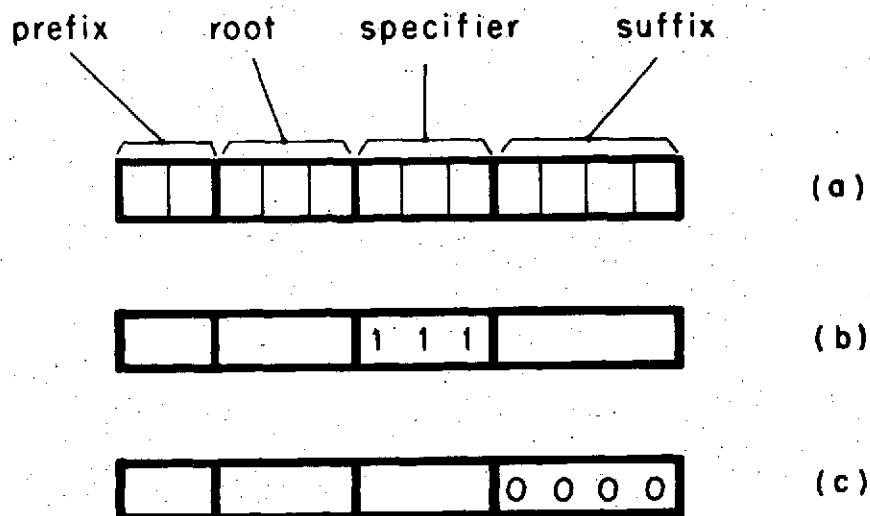


Fig. 21 - Morphology of words F in the CPL 1 processor

For the 12-bit words that describe configurations of the PN, the morphology represented in Fig. 21 is used. Type (a) is used for mode 1 of section 5.2.3. The root indicates the type of function involved, such as summation, subtraction, multiplication, etc. The prefix prescribes some details peculiar to each function, such as questions on overflow, sign, circularity, etc. The specifier specifies the arguments that are read for the data transformation, that is, it indicates the connectivity. For unary operations, the specifier specifies further details in the operations. Finally, the suffix indicates which of the four variables x_r are to be transformed.

For the individual prescriptions of mode 3 in section 5.2.3, the format (b) of Fig. 21 is used. The variable that is transformed is a preferred variable, and thus it does not need to be indicated. Root and prefix have the same role as before. The specifier has the pattern 111 for indicating mode 3 of the PN description. The suffix, in this use, refers to details and connectivity that are peculiar to each type of function.

For the special networks of mode 4, the format (c) is used. A pattern 0000 in the suffix (which is meaningless in the other modes) characterizes this format. The remaining part of the word is a coded list of special operating configurations. The prefix remains available for specifying options in each of these special configurations.

5.4 DISCUSSION

It is necessary to state in advance that comparisons between the programming language described in this report and conventional programming languages will always present some difficulty. For although in the broad sense of means of communication both are "languages", in several respects they pertain to different domains. The first produces descriptions of machines, the others descriptions of procedures. The first is in a multidimensional form and refers to a computer based on a programmable substratum; the second are in phrase structure form and refer to computers based on an instruction-obeying processor and random-access memory. Probably, the actual origin of this difficulty of comparison can be related to the repeatedly mentioned dual symbolic systems of mental processes, imagery and verbal structures.

However, because in both cases the final goal is the same, that of obtaining a certain result by means of a man-made machine, a discussion on the correspondences and divergences of the two types of languages should have meaning. In this sense, some elaborations on certain pragmatic aspects are made in the first section, and actual or apparent similarities with features of other languages are discussed in the following section.

5.4.1 Functional and technical characteristics

In conventional programming, the user description of the process and the actual execution of the computer are two different things. The difference is so fundamental that the user simply ignores what the computer does. However, what the computer obtains is completely determined (hopefully) by what the user prescribes. This determinateness is achieved by using a verbal-structure language very precise in its lexical and syntactical elements. This method on the one hand gives the user complete independence of the mechanics of the computer, and on the other hand constrains him to the rigidity of the grammar of that language.

In the CPL system, the machine does precisely what the user prescribes

(the important point is that it is not the user going to a given mechanics of the computer, but the computer coming up to the images of the user). Here determinateness is achieved by giving a structural description. A structure, a construct, appears to have larger latitude than a language of commands and declarations. The user describes directly "what" the computer should be and do. In this situation, the means for describing the "what" become almost irrelevant. Rather, essential is that user and computer work in similar spaces; this is made possible by the two isomorphic substrata: the symbolic one, used as user language, and the physical one, which is the computer.

From the experiences described in chapter 6, it does not result that a structural modeling, at the level of the symbolic substratum of chapter 3, is more difficult than a conventional procedural description. As a matter of fact, by pushing the user to visualize the processes structurally, and asking him to produce mental images, we help him to clarify the process and give him guidelines for developing a program.

Because the structures of the user language serve the user during the modeling of the intended process, and are not given directly to the computer, a large choice of vocabulary and a great deal of flexibility in lexical and syntactical characteristics can be accepted. Provision of the lexical rigidity needed by the computer is postponed to the time of the preparation of the machine program. Because of the one-to-one correspondance between the items of the user and machine languages, the preparation of the machine programs can be obtained, manually or automatically, without the involvement of processing activities.

Note that the use of relatively free notations does not preclude the writing of a precise, complete program in the form that the computer actually will execute. In conventional programming languages, if we use free notations (for the purpose of communication among human beings), it is difficult to define an actual computer program. The fact that a user language is relieved from lexical (and possibly syntactical) rigidity should be considered appropriate, as testified by the growing of default features in compilers.

There are several consequences from the approach taken, that only extended experience can fully clarify. Some of particular relevance are discussed in the following pages.

Machine independence

In conventional programming, we can distinguish several levels (often not neatly separated) each with a different degree of machine independence. At the so-called reference language level, a program is written unambiguously in the form of sentences obeying the vocabulary and the syntax of a particular programming language. At this level, a program is completely machine independent; it is potentially executable by any computer that has a compiler for that language and suitable system software. But, evidently, it is not an actual program that can be fed directly into a computer.

At the level of hardware implementations, a program has the same structure as above, but vocabulary and syntax are restricted to those accepted by a particular class of computer systems. Within that class of systems, such programs are complete and machine independent.

At the broad level of assembly languages, programs are expressed more in terms of features of a particular type of computer than in terms of a general language. Within that type of computer, they are actual programs and machine independent.

At the level of machine language, programs consist of the actual computer instructions and auxiliary control information. They are usually tailored to a particular installation. In conclusion, transferability is one objective of today's programming, but it cannot yet be considered achieved in a general sense.

In the CPL system, given the strong connection between user and computer, machine independence has to be viewed differently. Here we do not constrain ourselves to the automatic translation, but rather we attempt to optimize simultaneously the work of the user and of the machine.

The purpose of the user language is to produce a formal, rigorous, and complete description of a process, regardless of the actual machines that will eventually execute it. To give the user a chance to develop a good, essential description of the process, we eliminate actual machine constraints by implying ideal machines. However, these ideal machines have the same basic structure as the real machines, in order not to make difficult the later production of actual machine programs. This approach is made possible by using a language of abstract machines rather than a phrase structure

language. (Note that the high-level languages used today do not have the structure of the computers used today.) Thus, in the sense here discussed, the state diagrams of the user language are machine independent.

The machine programs constitute the actual information that is given a computer, thus they are fully machine dependent. In our case, the machine programs refer to a particular real substratum. But it is not the vocabulary that is relevant; given the one-to-one correspondence between the elements of these programs and the actual computer actions, it will always be possible to provide an automatic, direct translation from one vocabulary to another. What produces the machine dependency is the variety of the mapping (3.5) (section 3.2.2, page 80) that we can expect to be implemented in computers of this sort. Our approach is based on identity of syntax in the symbolic and physical substrata, but the mapping (3.5) has to be implemented differently, in different cases, for obvious reasons of cost and application. If these mappings are developed with a broad view, also without overlooking psychological considerations, we have a strong feeling that the different mappings can constitute hierarchical subsets of a general user language. In this case, a machine program for a CPL system of level k could be executable by all CPL systems of levels k and above.

Now we come to the connections between user and machine languages. No particular attention to this subject has been needed in the present work. In the use of the CPL 1 machine, the state diagrams are sketched initially in a summarized form; then they are redrafted with the actual capabilities of the CPL 1 machine in mind. Then, the machine programs are hand produced, by writing the items of the state diagram on punch cards, in terms of the words of the dictionary (part of the manual of the CPL 1 machine). Because of their meaningful morphology, most of these words are mentally composed when punching the cards, without reference to the dictionary. When a program is complex in terms of the capability of the machine used, a preliminary, careful check of the implementability of the state diagram is strongly advisable. When a program is simple, details and variations can successfully be improvised directly in machine language.

In chapter 6, programs are shown at different levels of the user language (state diagrams), and in machine language for the CPL 1 processor, that is, on punch cards. The user languages ranges from the extended

implementation of section 5.2 to the subset implemented in the CPL 1 processor. Moreover, for the sake of clarity, programs that were developed originally for the CPL 1 machine are also often presented in the user language of section 5.2

In conclusion, we can depict the situation as follows: A user program (state diagram or equivalent) is a formal description of a process that can easily be read and interpreted. It is a complete program. It is executable by CPL machines above a certain level, and not by other CPL machines. In the first case, the machine programs are obtained by transliteration of the graphic and alphanumerical symbols of the state diagram into the corresponding machine words. In the second case, the programs should be re-elaborated before the transliteration can be made.

For a larger context, the following conjectures are offered. User languages could be developed with lexical and syntactical characteristics as appropriate for each application, but always isomorphic with a basic, general, symbolic substratum. Programs written in these languages could be translated by automatic or manual procedures into the terms of a standard vocabulary and in a standard FSM form of a desired level k . Each of these user languages will probably be translated automatically down only to a certain level k . The productions of these translations could be considered actual machine programs, for machines of that level, because every nonnaked CPL computer will have incorporated direct translators from that standard form and vocabulary.

In order to permit the one-to-one correspondence between the elements of the user and machine programs, it is necessary either for the physical substratum to have the same features as the symbolic substratum, or for the symbolic substratum to be limited to the features of the physical substratum. The first course would be adopted when possible, otherwise the second would be. The interesting characteristic of this approach is that each machine language can be viewed as a restricted form of the user language, rather than as a completely different language. When a derivation is desired between a program and a CPL machine not isomorphic with the language used in that program, a sort of compiler could be developed, or human intervention used. In this case, the work of adaptation should not be impractical because all machine languages would have the same structure as the programming languages.

In this report, the possible connection of a CPL system with present programming languages and computers is not analyzed.

Data and declarations

One of the significant divergences from conventional computers is in the treatment of data. In conventional computers, data are individually piled in a memory, more or less randomly, and are referenced from and to it by means of an address apparatus. Here data are created in structures by the operational substratum PN, and, in a virtual sense, they remain in that form (the pages are virtual replicas of PN). A first consequence is that here the address apparatus is unnecessary.

In a random-access storage system, the specific random locations of the single variables are, obviously, without any interest. Accordingly, in programming languages data are referred to by names, which constitute a much more natural reference system. However, this method generates a significant overhead in the work of the computer, and often creates unsuspected pitfalls for the user (such as in calling by name or by value). In the CPL system, data are grouped in pages, and the pages are usually not called for, but are associated in further structures (sequences, arrays, matrices, etc.) so that data emerge spontaneously at the proper places and right times. This organization is made with ease directly by the user because programs are developed in the form of abstract machines, rather than as sequences of phrases. If the storage substratum PM has enough flexibility, the user can mold in it the images he has of the data structures.

"The simplest kind of structures for us to verbalize and work with are spatial in character. It is not surprising, therefore, that there are many examples of use of n-dimensional space." (Miller 1964, p. 224). In accordance the memory is organized as n-dimensional structures rather than as an unstructured deposit. Here the user "sees" the actual structure and evolution of the computer, and we can therefore expect that he will be less error prone than when he is dealing with collections of items indirectly, through a verbal language.

The passing of parameters to a subroutine, procedure, or function here occurs as an interplay between FSMs and pages. The pages choose portions

5.4.1

of programs by means of the key word, and the program, in turn, transforms the pages and their key words. Undoubtedly a large flexibility is available for constructing complex processes.

One might suspect that a complex interplay of many pages with many FSMs would generate confusion in the user. In the context of the processes analyzed in this report, it does not appear to be so. The conjecture is made that, for large classes of processes, the geometrical intuition that the user can develop for the structures that he designs in the framework of Fig. 17 is no less effective than the symbolic intuition he can develop for the structures available in the virtual machines created by the compilers. For the classes of problems in which a verbal structure is the appropriate model, such structures can be implemented in the frame of Fig. 17 with no more difficulty as when they are implemented by a compiler in the substratum of conventional computers.

A different approach results also in regard to the declarations. Because data are not called for but are constructed and manipulated by the operational substratum, there will be no declarations attached to the data. Rather, the operational structures have specified how to treat data. This permits different treatments at different times on the same data, without the need to introduce new declarations.

The notion of declaration has undoubtedly a great psychological beauty, in the sense that one can start by saying "be such and such...", and all the necessary context and background comes into being. Unfortunately, in present programming languages little of this potential is implemented, and the largest role of declarations is devoted to mechanical details required in a random access system. In common communication between persons we do not need to declare those detail for every object; in most cases that information is already implicit in our discourse; only in particular instances we take recourse to an explicit declaration. In this respect, the declarations of conventional programming languages are certainly annoying, and undoubtedly redundant.

In assigning the prescription of data treatments to the operational structures, rather than to the data, we have some advantages. When the user is preparing transformations on certain data, he knows what those data are, and evidently he does not need to make a declaration to himself. Thus, if

later he wishes to treat those same data differently, he can do so and again does not need to insert a new declaration. All the problems of validity and consistency of declarations in presence of transfers, go to, call, etc., do not appear here. Undoubtedly it would be difficult to implement with compilers the flexibility achieved when the user develops directly both the data and operational structures. When the structure of a process is visible to the user in the form of an abstract machine, with all the dynamic aspects made clear by a spatiotemporal frame, the user is in much better position for exploiting his capabilities in creating an efficient program.

It is interesting to note that here data and program are in one respect more separated and in another respect more embedded than in conventional computers. The pages (data) and the FSM descriptions (program) follow completely different paths. However, when they merge together in the programable network, they become indistinguishable; the structure of the page is part of the operating structure, and the operations create the structure of the pages.

Efficiency of programs

The efficiency achievable by hand coding in conventional computers is well known, but it can rarely be afforded because of the labor required. The simplifications that can be made in the programs when new hardware features become available are also well known, but they would require the purchase of a new computer.

Here, because of the two isomorphic substrata, we have a machine language that is at the same level as the user language, thus the user can always make the proper choices for the computer execution. Moreover, the operational substratum allows the implementation of all sort of hardware features.

Ease of entering and modifying a program

Conventional programs are expressed in the form of a listing. It is well known the difficulty of recovering and understanding a process from the listing for anyone who did not write that program. The reason is that a

listing is a very unexpressive form of representation for a process, and requires a great many symbolic steps before it can be reinterpreted. A confirmation of this fact is the use of comments as an almost indispensable ingredient of programs.

An abstract machine represented in the form of a state diagram is, on the contrary, a very expressive object. Its graphic structure easily evokes the spatial intuition. The symbols embedded in it allow an easy understanding of the dynamics of the process through anticipation or temporal intuition. The modularity in terms of pages and states keeps programs understandable regardless of their complexity. The hierarchical description in terms of FSMs, states, networks, and prescriptions allows a gradual acquisition from an overall picture to the minute details.

For similar reasons, modifications of a program are much simpler when one deals with a state diagram than with a listing. The modularity of the states allows an easy development of new parts. The absence of declarations eliminates one common source of trouble. The easy visualization of the entire process given by the state diagram permits a careful verification of all consequences produced by each modification.

The interpretation of conventional programs is based on a foreign grammar that needs to be memorized, and thus is prone to errors. A typical example is the extended use of concatenated parentheses in certain programming languages; parentheses are a very concise, elegant, and effective device for symbolic representations, but they become inappropriate for mental processes when extended excessively. A CPL program, instead, is largely based on visual structures for which we all have a special facility, and which are less error prone. The modeling in the form of a machine allows the intuitive understanding of very complex dynamics.

Interaction with the computer

The drastic difference in the ease of man-machine interaction between the CPL system and conventional computers should be evident, and little more needs to be said. The basic reason is that in conventional computers communication occurs through verbal-like messages, such as commands, statements, and declarations; in the CPL system, the user constructs and sees continuously

the machine. This approach, compared to conventional programming, can be paralleled to "doing it yourself" compared to telling some simple-minded person to do it. In the latter case, you have first to model into verbal structures what you had in mind, anticipating the actions he can make; then, you have to be sure that your listener interprets correctly your utterances, for the same result you wanted. If you do it yourself, instead, you map your images directly into actions; and this is particularly direct, because in conceiving your intentions you were anticipating, more or less unconsciously, what you were able to do.

Because this is an unusual viewpoint in modern computers (while it was not unfamiliar at the early time of computer development, especially the analog ones), another analogy is offered. In a factory, if people were communicating only by telephone, in different languages through interpreters, without blueprints and personal contacts, the work would be still possible but more limited and difficult. Each method of communication can suffice for transmitting all sorts of information, if suitable means of translation are provided; but when many different types of information are involved, communication is more efficient if more than one method is used.

Debugging

The fact that here the computer assumes the configuration of the model chosen by the user for his problem makes debugging a part of understanding the problem, rather than an extraneous activity. The fact that user and computer work in two substrata that are isomorphic eliminates many difficulties typical of the cases in which user language and computer actions bear no similarity.

Diagnostic facilities are easy to be implemented due to the fact that data are organized in accordance with the structure of the problem, and the execution proceeds in the modular frame of the states. Many of the comments made in regard to the ease of entering a program apply also to the ease of debugging. In the CPL system, a direct machine dump produces the variables x_i partitioned in pages and states. Such a data display is completely meaningful for the user, because those variables, pages, and states are the elements into which the user modeled his process. No debugging facilities are necessary in the language.

5.4.2 Similarities with other programming languages

Flowcharting

Flowcharting started with the "We therefore propose to begin the planning . . . by . . . the flow diagram . . ." of Goldstine and von Neumann (1947). The first computer programs were flow diagrams with an accompanying list of codes. Then computers developed, and today programmers do not use flowcharts, except as a secondary, simplified documentation coming after the program has been written. Conversely, in programming the CPL system, the first step is the production of a state diagram, which is related to the early flow diagrams in several respects.

The reason for all this is very simple. A graphical representation of the type of a flowchart is the most effective description of the behavior of a processing machine. At the early times, programmers were dealing directly with the actual actions of the computer; thus flowcharts were the most effective program representation. Then programmers freed themselves from the actual working of computers and dealt with problems in terms of phrase languages; in this case, flowcharts are completely useless. Our FSM is again a machine, and the programmer deals directly with it, actually conceives and constructs it; therefore a type of flowchart becomes again the most effective program representation. The difference is that in the early times of computers, the flowchart was describing the behavior of mechanical or electronic devices that exhibited no resemblance with the problem as seen by the user; here the state diagram describes the behavior of an abstract machine that is the model of the problem as seen by the user. Once this situation is achieved, it is not difficult to enrich the flowchart with a variety of features and notations to make it a very expressive, user-oriented programming language.

The power of expression of a graphical representation is so obvious that program theorists often were intrigued with it, aside from its use for representing the actual work of the computer. It has been proposed to use flowcharts not only as a notation but even as a programming language (Burkhardt 1965); however, it never has been adopted. Galler and Perlis (1970), in a general analysis of programming languages, raise the question: "The clarity and precision we have achieved in representing algorithms by means of flowcharts leads one to ask what it is about flowcharts that makes them so much clearer than the verbal description". They recognize that it is not the two-dimensionality of the representation, because any flowchart can be easily

transformed into a linear sequence, but they do not elaborate further.

In our interpretation, we recognize that the effectiveness of a graphical representation has a psychological basis. A graphical representation of an abstract mechanism directly evokes the imagery system of our mental processes. Instead, a sequence of symbols, conveying the same information, needs first to be memorized in its entirety (and that implies an effort), then it is analyzed, and finally the mental images start to take shape. In the case of a program, we can say: when an algorithm is represented in a list form, what we perceive first as a unit is a single command, or declaration, and it requires a certain amount of work to reconstruct the entire mechanism from the many commands. When the algorithm is represented in state diagram form, what we perceive first as a unit is the entire mechanism, and we then need little effort to focus on the single parts in order to make precise our knowledge of the algorithm.

It is also interesting to see a similarity in the recommended approach to programing. Here we have two successive phases: first, the delineation of a strategy, a mechanization, in a relatively free user language; second, the elaboration of actual machine operations. For the first computers, we read that "it is advisable to plan first the course of the process and the relationship of its successive stages . . . and to extract from this . . . the codes . . . as a secondary operation" (Goldstine and von Neumann 1947). However, when programing is made with a phrase language in which lexical and syntactical characteristics are rigid and crucial, such a gradual approach is not very practical. The reason for this is that visualization of a process in its entirety requires remodeling it in a context that is different from that of the phrase programing language; thus we have to switch alternatively between a visual frame, and a verbal frame, and attempt to establish connections between the two.

Elaborating further on the above comment, we can realize that the state diagrams of our abstract machines, while having a similarity of roles with the flow diagrams of early computers (the main differences being in the level of the modeling and in the richness of features), have an intrinsic difference with the flowcharts of today's programs. This difference comes not from the different number and types of boxes, but from a more subtle reason. Flowcharts are graphical representations of processes already modeled in a

phrase structure; thus, they can enjoy the fast perception peculiar to graphical representations, as it concerns the overall picture, but they are bounded to the features of a phrase language for what concerns the process modeling. Instead, state diagrams are the actual model of the process, made with full use of the features peculiar to the images of our mental processes (compare with the game of charades discussed in section 2.1.4). A further difference is in the fact that flowcharts are a model for the user, but not, typically, a representation of the actual computer execution; state diagrams are both.

Ironically, flowcharts are highly recommended in the programming courses, in recognition of the power of graphical representations, but professional programmers find it preferable not to use them. Only seldom, for an intriguing process, a programmer may start with a graphical sketch of the overall structure of the program, but then, at a certain point, he transfers to phrase structures. A graphical representation, if carried out to the actual details of a program, becomes too cumbersome, thus useless. If the flowchart is kept at a level in which it can function as a panoramic image, it does not constitute an actual program. In practice, flowcharts are used only as an auxiliary, summarizing documentation of programs already developed.

The power of the state diagrams derives from modeling the processes in the form of automata (our abstract machines). The inconveniences that arise in flowcharts are not present here, because graphical and verbal means are used in a collaborative way, and not each as a different representation of the other.

Karp and Miller (1967), Slutz (1968), and others have applied flow diagrams to the analysis of parallel computation. They need to distinguish data operations and control structures, because they imply conventional computers. Here we imply a computer isomorphic to the programming language, and it is possible to deal with only one structure that is representative of both the data transformations and the control.

APL

Among the programming languages that have been developed, APL (Iverson 1962a) is one that suggests more similarities in structure and goals with the language here described. However, the similarity is more in the appearance because the two languages are in different frames.

First of all, more than a language for actually programming computers,

APL is a system of concise and powerful notations applicable to a variety of descriptions and analyses. In particular, it can be used for describing a process for a computer if suitable compiler, or interpretive routines, are available (Falkoff and Iverson 1967). APL per se does not provide data structuring, input and output, which are crucial points in computer use. Works of direct implementations were cited in section 1.3. The language described here, instead, is a method for actually programing a particular type of computer, the CPL system.

In APL, the emphasis is placed on conciseness in describing algorithms; this is inevitably paid for with a compulsory notation system that is difficult for the nonexpert. Consideration to computers appears only in the structure as sequences of statements, clearly motivated by the hope that simple interpretive routines will be able to apply the languages to the different computers. Because these sequences often are very concise and use arrows for jumps, APL programs have some diagrammatical appearance, thus resembling structures of FSMs. However, there is complete absence of the notion of state, which is crucial both for visualizing the structure of a given process, and for gradually constructing a complex program. The viewpoint is fully that of sequence of statements.

The point is that APL reaches conciseness by means of elegant notations and not becoming involved in the execution. Iverson himself (1962a) says that the goal actually is to provide a language with such a descriptive and analytical power as to repay the effort required for its mastery. He shows (Iverson 1964) the interesting analytical possibility of the language. In a sense, programing a computer is incidental. The language described here on the contrary reaches conciseness by prescribing an execution that is tailored to the model chosen by the user for visualizing the process. The notation to be used is not of primary relevance for the method. We see that the scopes of the two languages are in different areas.

Iverson talks of common language for hardware, software, and applications (Iverson 1962b). He recognizes the desirability for a programmer to deal with a process at a high level, while having the possibility to specify also details. But what he means is that APL is very effective (concise) for describing and analyzing the working of a given piece of hardware, for describing and manipulating a given piece of software, and for expressing a

given algorithm. All this is very valuable but it is different from the goal expressed in section 1.1 of merging the three points of Fig. 1. We are not interested in describing hardware, software, or algorithms per se and separately. In our approach, hardware, software, and algorithms are all aspects of the same structure. In this sense not only does a single language describe them, but it is actually a single description. For APL, there are three different descriptions for the three aspects. This different situation is a consequence of the fact that Iverson worked only on the language, accepting the computers as they are. Here the computer also is re-examined and changed together with the language.

Decision Tables

For more than ten years (Kavanagh 1960; McDaniel 1970; Low 1973) decision tables have been recognized to be a very effective programming method, easily understood by humans regardless of their background, and machine independent. One item of the FSM description, the function T, has in a sense the same philosophy of decision tables. Therefore similar advantages can be expected.

One difference is that conventional decision tables need to be compiled, in order to be understood by a computer. Here function T is directly implemented by the loose hardware of the CPL system. Another difference is that common decision tables need some interface with the other general purpose programming languages. Here function T is one of the constituents of the language, therefore it is well integrated in all kinds of programs.

C P L

A programming language has been developed in Cambridge, England, from which the name Cambridge Programming Language (CPL) is derived. This language is to be used with conventional computers, but has a particular aim to be efficient also for nonnumerical processes (Barron and Strachey 1966). In this context, the language is advantageously structured in blocks. However, it is a phrase structure language. There are similarities of aims and orientations in the Cambridge language and in the language of this report, however only the names are identical.

Chapter 6

Comparison of Programs

The main objective of this work, as described in the previous chapters, is to provide a more direct communication between man and computer by introducing two corresponding substrata, one symbolic, which has the role of the programming language, and one physical, which constitutes the computer. If the symbolic substratum is suitable for representing the "images" that the user conceives for his processes, and the physical substratum is isomorphic to the symbolic one in such a way as to make those images real, we can expect a greater facility for making computers do what we want them to do.

The purpose of this chapter is to test to what extent this expectation can actually be realized. Moreover, we see in practice that each programming language (and in some degree each type of computer) is more efficient for some classes of problems and less so for others. If the programming language and the computer have the greater versatility of substrata where varieties of "abstract machines" can be developed that conform to the specific processes, we may expect also a more uniform efficiency in programming and execution for the different types of processes. To test also such a possible uniformity, sample programs for processes of different natures will be considered.

6.1 INTRODUCTION

6.1.1 Available works on program comparison

Comparing computer programs involves two different classes of considerations. On the one hand, programs are computations, and as such they can, in principle, be approached with mathematical theories of computation and of its complexity. On the other hand, programs are affected by the technical charac-

teristics of the specific computers for which they are prepared. These characteristics do not follow, at present, organized criteria, but rather they are the consequence of empirical and economical considerations. The increasing relative cost of programming (see for instance Balzer 1973) has promoted numerous studies for increasing the software productivity by means of tools derived from theoretical considerations. However, at the present time, no practical procedures are available for handling simultaneously the theoretical and the practical aspects of a program.

On the theoretical side, among the studies that can be closely related to the building of theoretical tools for program comparisons are the works of Chaitin (1966) on the length of programs for Turing machines, the intriguing properties determined by Blum (1967) on the number of steps for computing functions, and the studies of Meyer and Fisher (1971) on the succinctness of different forms of descriptions.

On the practical side, only pragmatic considerations can be found, such as the comparisons of Schwartz (1965), and the discussions of Shaw (1966). An attempt to establish a criterium of measure is in Hellerman (1972). Crucial in any comparison of programs is their rigorousness, and the works of Dijkstra (1968, 1972) also become pertinent.

6.1.2 - Criteria of comparison

In the absence of any developed criterion for evaluating and comparing programs, only empirical parameters can be considered and pragmatic characteristics of the programming language evaluated.

Attempts to determine some significant parameters for comparing actual programs were all nullified by the differences between conventional languages and the language of the abstract machines. Thus, simple, external data such as the number of statements, symbols, cards, and memory bytes have been accounted. When possible, the execution time and the programming time were also measured. The interpretation of these data is given in the next section.

In regard to the comparison of pragmatic characteristics of the language, a qualitative account is given here, under the titles that usually appear in the literature.

Ease of learning. We have here to distinguish two types of users: those who have not previously been exposed to computers, or who have had only sporadic contact with them; and those who have routinely worked with conventional computers. For the first category, no difficulty has been encountered. It was interesting to observe the benefit of the physicalization of the processes in the form of abstract machines. For the second category, there is indeed an initial phase of changing habits. After this, of course, there is the benefit of more competence.

Elapsed time for programing. Given the complexity of producing a reliable measure of the actual time devoted to a program, few measures appear on the tables. However, a shorter programing time is one of the most significant results of these comparisons. If we compare at the machine language level, the difference is between one and two orders of magnitude. If we compare the time for writing a program for the CPL system (machine program) with the time for writing an equivalent program in a high-level conventional language, there is no significant difference in general. But if in the comparison we include also the time for making the program work, a great difference appears in favor of the CPL system.

Ease of debugging. In agreement with what can easily be expected (see page 167), the phase of debugging for the CPL system is either of negligible importance or very simple to carry out.

Program readability. "My opinion is that programs with more than a certain number of steps are absolutely unreadable, no matter what language they are written in." (Shaw 1966). This is a common comment about conventional programs. The experience with the programs written for the CPL system is quite different. The state diagrams, as shown in the following sections, do not lose their clarity with increase of size of the program. Given the ease of writing programs, often punch cards of the CPL machines would accumulate disconnected from any reference; it was possible to reconstruct the process by reconstructing the FSMs even from the punch cards.

Ease of documentation. In conventional languages, a good documentation requires an extra effort on the part of the programmer. In the CPL system, the state diagram, as produced during the development of a program, constitutes

an objective documentation, that includes all the information that may be needed in the future for understanding, maintaining and modifying a program.

Ease of maintaining and change. The experience obtained so far with the CPL system has been all in a context in which extensions and modifications to the programs were normal events. The facility for documenting the programs, the readability of the state diagrams, and the ease of debugging discussed above are all characteristics that permitted the maintenance of programs to be made without difficulties.

Several hundreds programs of different types and scopes have been written for the CPL machine during these past years. Although all this material could not be specifically documented, even in a statistical form, these programs constitute the basis upon which the various comments, discussions, and statements have been made throughout this report. A few sample programs have been selected for the comparisons, either because of the interest in the processes performed, or because more documentation was readily available in the corresponding conventional programs. Most of the sample programs are in the context of real-time processing of weather-radar signals, and they are grouped in section 6.2. Some of the programs are from the early use of the processor at the Radio Meteor Project of the Smithsonian Astrophysical Observatory, and these are grouped in section 6.3. Section 6.4 shows some exploratory programs developed in a theoretical context for analysing the effectiveness of the system in different applications.

Acknowledgement is given to the several people who at different times and places, contributed to the program comparison, either by running the CPL machine, or by developing CPL programs, or simply by writing corresponding programs in conventional languages. I am thankful to David Hallenback, Julian Simms, Steve Tubbs, Robert J. Horn III, Ken Yeager, Hon W. Chin, Jerry Morrison, and Man Kong Yau.

6.1.3 - Description of the data used in the tables

The data that appear in the tables of comparison in the following sections are to be interpreted as follows.

Language. The language in which a program is written, ranging from high-level languages to machine codes, is referred to in its common name.

The name CPL 1 is used for the programs written in the language of the CPL 1 processor described in section 4.4.

The name CPL 2 is used for programs written in a more extended language that is expected to be implemented in a CPL 2 machine whose construction has been initiated. This language basically corresponds to the level described in section 5.2.

Statements/Instructions. These terms have conventional meaning when they refer to the conventional programs. In the CPL programs, each independent expression that appears in the state diagram is counted as one statement; e.g., an expression that describes a configuration in F or an expression that describes a condition with its related path in T.

Symbols. Each independent component in an expression is counted as one symbol; e.g., a name, a command, an arithmetic symbol, a significant delimiter, punctuation mark, or blank. Also in the CPL language each graphic delimiter for distinguishing states, F, T, and R is counted as a symbol. Moreover, each word describing a configuration is accounted for as many symbols as many bytes are necessary for its storage.

Number of cards. The cards considered are those employed in each language. For high-level languages, usually there is one card per statement. For machine languages, often several instructions are written in the same card. For the CPL 1 programs, the format of the card is the one described in section 5.3.2.

Memory words. This is the number of words of memory that are used for storing the program. If not otherwise specified, words of 12 bits are considered.

Execution time. In each case, a portion of processing is chosen such that it is representative of the total execution time, as well as being suitable of measurement. In some cases, the fact that the process is executed in actual real time is indicated with r.t. The indication q.r.t. (quasi real time) is used in cases in which the actual execution occurs after a preliminary storage of the data in the memory, but, for all practical purposes, the results appear as if they were obtained in real time.

6.2 REAL-TIME PROCESSING OF WEATHER-RADAR SIGNALS

Radars have been used for the observation of meteorological events ever since World War II. Special radars are now being gradually developed as specific meteorological instruments. The development of this field can be traced in the records of the Weather Radar, and Radar Meteorology Conferences of the American Meteorological Society (Boston, Mass.).

As eyes constitute a powerful sensor for biological creatures insofar as they are closely connected to a sophisticated processing system (the brain), similarly, the degree to which radars applied to meteorology can produce information of meteorological relevance can be thought of as being related to the processing facility they incorporate. The most extensive experience with the CPL system has been carried out in this context, from 1970 to date, at the Massachusetts Institute of Technology. In this field of application, the CPL system appears to be particularly appropriate for the following reasons:

(1) For radar meteorologists, computers are only an auxiliary instrument, and not an object of primary interest. Therefore, a solution is highly desirable for which the wanted processings can be prepared with the least effort, and without the need for professional programmers.

(2) Weather radars produce a very high rate of data, all of which may contribute to the information of interest. Therefore, a system that can process in real time a large quantity of data without a large investment in equipment is paramount.

(3) Very little is known about the types of processings appropriate for extracting the information of interest from the raw radar signals, and years of study and development are expected. In this situation, for economical reasons, an approach is appropriate that is the least committed to a particular solution, and that allows for the greatest flexibility and interaction.

In the following sections, samples of the programs that have been developed in this field are described, and compared with equivalent programs for other types of computers. The programs were prepared for and run in the CPL 1 machine; for some of them the more concise versions written for the CPL 2 machine, or written in the language of section 5.2, are shown. Some proposed programs directly developed in the language of section 5.2 are also included.

In order to give an idea of the practical advantages that can be derived from a large availability of real-time processing, a few samples of different types of results are shown in the following figures.

Fig. 22 shows that inferences about types of precipitation can be drawn from a characterization of weather echo patterns made on the basis of some structural characteristics (program described in 6.2.1). The interest for such a processing is in having that information at the same time as observing the echo patterns on the radar screens.

Fig. 23 shows a particular evolution of echo pattern simulated with a numerical model, from an initial simple cell. Numerical models of this type (described in 6.2.10), fed with initial data collected in real time by the radar, are of possible interest for developing a short-term forecast of the evolution of the echo patterns, and for inferring from these evolutions the meteorological events that might occur.

Fig. 24 shows an example of elimination of ground echoes in the data collection of a weather radar; in this instance, the echoes from the Monadnock Mountains (southern N.H.) are canceled in real time during a PPI recording. In frequent geographical situations, ground echoes strongly disturb the radar observation and measurements of meteorological events. However, ground echoes exhibit a spectrum of fluctuations significantly different from that exhibited by weather echoes, and thus they can be recognized and eliminated with a suitable processing in real time (program described in 6.2.4).

Fig. 25 shows a measurement of the radar antenna pattern obtained with the solar noise. For quantitative measurement of precipitation by radar, a precise knowledge of the radiation pattern of the antenna is required. Measurements of this sort are always delicate and expensive because they require targets remote from the ground, such as high towers, an airplane, etc. The sun is available in every part of the world, it is well isolated in the sky, and it produces a broad noise spectrum. If an appropriate processing is made to enhance the intensity of the noise and to overcome the short-term variations (program described in 6.2.8), and the distribution on the solar surface is accounted for, practical measurements of the antenna patterns can be obtained with simple means.

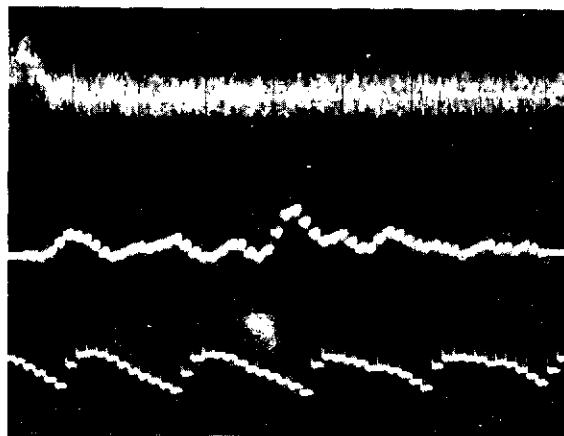
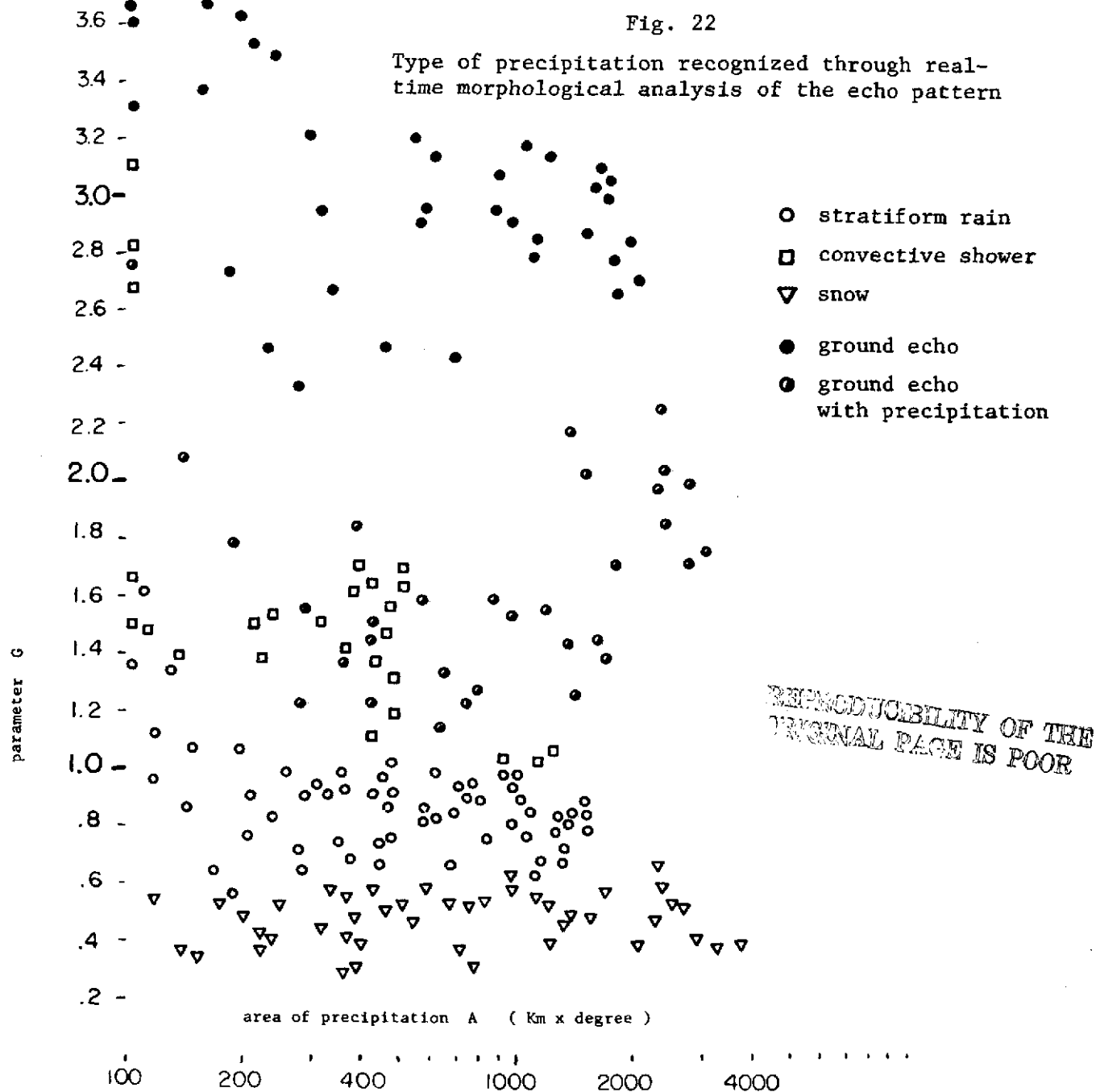


Fig. 23

Different evolutions from a single cell simulated with a real-time numerical model

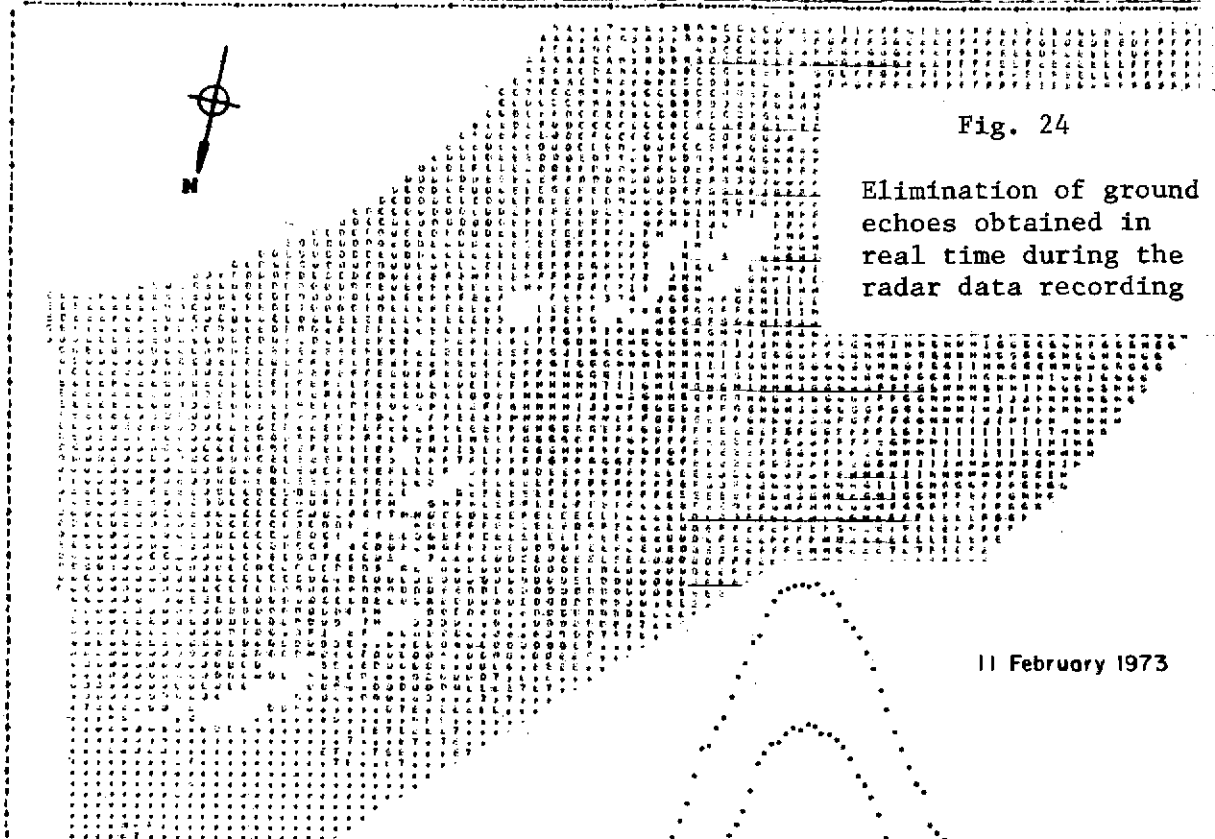


Fig. 24

Elimination of ground echoes obtained in real time during the radar data recording

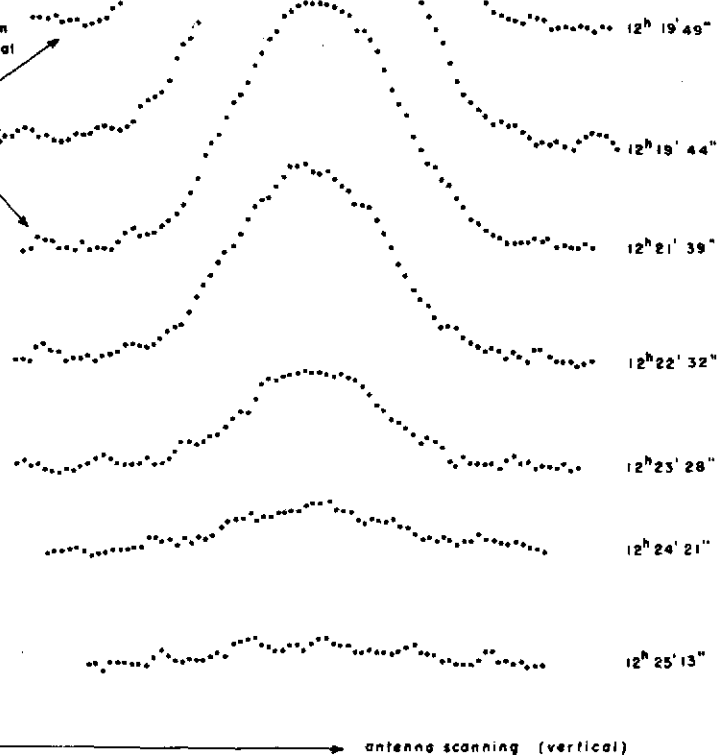
11 February 1973

Fig. 25

Antenna pattern determined by means of the solar noise

profiles correspond to a different position of the sun. (horizontal movement.)

output
(integration of 10^3 samples every $\frac{1}{2}$ second)



REPRODUCIBILITY OF THE ORIGINAL PAGE IS POOR

6.2.1 Structural characterization of radar-echo patterns

The structure of radar echo patterns, obviously, bears some relation to the characteristics of the meteorological events of which they are the radar visualization. What would be helpful is the determination of some parameters, easy to compute on the radar signals, that could reflect, in a global, statistical, concise form, characteristics of interest.

The program to be described processes in real time the radar output during the horizontal antenna rotation, and produces three global parameters related to the structure of the echo patterns. Echo intensity $x_{i,j}$ is considered in two discrete coordinates, the range i , and the azimuth j . The raw echoes emerge from the radar in sequences of successive range points i . These sequences correspond in time to successive azimuths j , due to the rotation of the antenna.

Because of the fluctuation characteristic of weather echoes, the echo intensity is determined, at each point, as the mean value of the raw echoes in 32 successive sequences z :

$$x_{i,j} = \frac{1}{32} \sum_{z=n}^{z=n+32} x_{i,j,z} \quad (6.1)$$

The following global parameters are determined for a given area explored in the atmosphere.

area of precipitation = α = the number of points i,j at which the
the echo intensity is above a threshold t

$$\text{mean value} = M = \frac{\sum_i \sum_j x_{i,j}}{\alpha} \quad x_{i,j} > t \quad (6.2)$$

$$G = \frac{1}{2} \left[\frac{\sum_i \sum_j |x_{i,j} - x_{i-1,j}|}{\alpha} + \frac{\sum_i \sum_j |x_{i,j} - x_{i,j-1}|}{\alpha} \right] \quad (6.3)$$

The G parameter can be considered as the mean value of two terms thought of as a radial and azimuthal computations. The radial computation can be visualized with the aid of Fig. 26. By computing expression (6.1) at all points, an echo profile A is determined; a replica of it is shifted by one range point; then the absolute value of the difference between profile A and B is computed at each point. The sum of these differences for the entire area, divided by α , is the radial computation. The tangential computation is made

in a similar way, except that the differences are made between profiles that succeed in time, which correspond to adjacent azimuths.

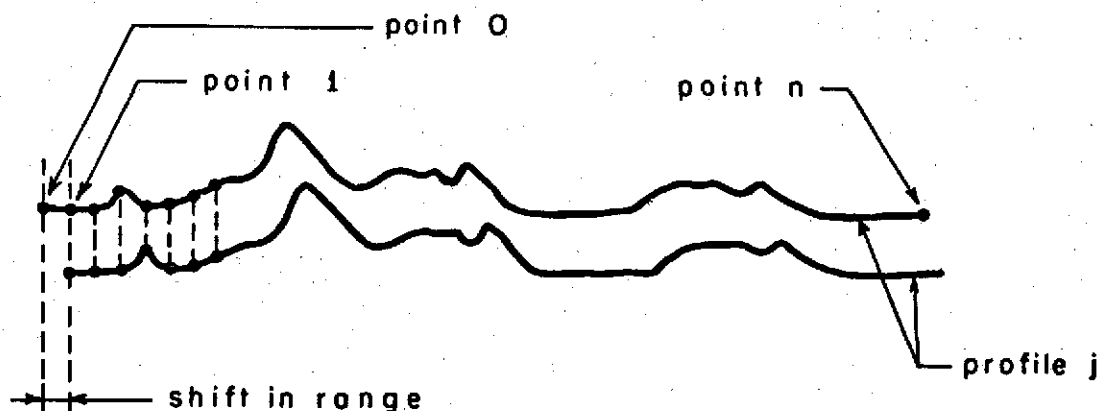


Fig. 26 - Echo profiles versus range

In the following an implementation of this process by part of a CPL machine is shown.

This first example is described in detail, to give the opportunity of seeing the language of section 5.2 applied to an actual program. In the following examples, the features explained here will be considered familiar, while each new feature will be described in detail.

No preliminary storage of data is made. Rather an organization of pages is chosen such that data are processed directly as they arrive. The circulation of the pages is made synchronous with the periodicity of the radar, and a sequence of pages is allocated in correspondence to the range interval of interest. Thus the computations at all the n range points can proceed simultaneously in the corresponding pages.

Then, an abstract machine is devised composed of four FSMs (Fig. 27). FSMs 1, 2, and 4 are implemented by one page each, and FSM 3 is implemented by the sequence of n pages corresponding to the n range points. FSM 1 first creates the organization of the pages, and then controls their work; FSM 2 provides the echo intensity at the initial point (0 in Fig. 26); FSM 3 describes the computation at all subsequent points of the profile; and FSM 4

provides for the preparation of the global parameters.

Description of FSM 1 (control)

For starting the program, a page is introduced into the system (solid arrow) in FSM 1, state 1. Here, variable A increments by one at each cycle (function F). After the first cycle, the condition $A = 1$ (transition function T) is valid, thus the page remains in PN (circle on the arrow), and creates a page in FSM 2 (routing function). Then for n consecutive cycles, the condition $A < n+2$ is valid, and n pages in FSM 3 are created. At the subsequent cycle, the third condition is valid, and one page in FSM 4 is created. At the next cycle, no condition is valid and the page transfers to state 2, this time leaving the circulation of all pages free (no circle in the arrow).

The transition to state 2 has a notch with the number 31; this means that the page will remain idle for 31 circulations, before reaching state 2 (transition i in Fig. 21). In state 2, the page produces a driven transition to state 2 for all the pages in FSM 2 and FSM 3. Subsequently, the page goes to state 3, after 31 idle circulations. The reason for this state is to disregard the first tangential computation, which would refer to an initial all-zero profile; this is obtained by avoiding the driven transition to FSM 4.

In state 3, the page produces a driven transition to state 2 for all pages of all FSMs, with a periodicity of 32 circulations (31 idle circulations plus one in state 3). At the same time, auxiliary variables x' are cleared. This continues until an outside control signal END appears; at which occurrence, the page transfers to state 4.

In state 4, the page produces a driven transition to state 3 for all pages, acquires auxiliary data such as azimuth, elevation, the threshold t used, and messages, routes these data to the output buffer, orders a record (or a printout) of the entire content of the output buffer, and then the page disappears (triangle).

Description of FSM 2 (initial point)

As a consequence of the driven transitions produced by FSM 1, this page accumulates 32 video samples s into variable A (state 1), divides this accumulation by 32 (state 2) and transfers the obtained value (the mean echo intensity at that point) into variable B' (in the auxiliary page array Ω').

Variable A assumes the value of the present sample, in state 2, for starting a new accumulation. FSM 2 simply computes expression (6.1) at the initial point of the profile.

Description of FSM 3 (points of the profile)

Accumulation of 32 samples is provided as in FSM 2. But here, in state 2, the mean echo intensity $x_{i,j}$ obtained in A is copied into variables B, C, and D (second expression). Then variable B is shifted one position along the pages, through interchange with B' (third expression).

Now, C and D contain the local echo intensity $x_{i,j}$; B contains, as said, $x_{i-1,j}$, the intensity at the point adjacent in range; and E contains $x_{i,j-1}$, the intensity previously determined (as it can easily be traced from the sequence of expressions and cycles) which belong to the adjacent azimuth. With the fourth expression, we produce in C and E the absolute value of the differences between C and B, and E and D, respectively. Thus C and E now contain the terms under the summations in expression (6.3).

The fifth expression accumulates into the auxiliary variables C', E', and D' the above values in C and E, and the present $x_{i,j}$ which is still in D. Finally, if the present intensity (variable D) is larger than a threshold t , an increment of one is routed to the outside variable $\delta(o)$. This is for computing the area of precipitation.

The reason for making such a parallel computation (which may appear more intriguing than a conventional serial algorithm) is to obtain the entire computation in real time, for all the adjacent points, without losing any single radar echo.

Description of FSM 4 (global parameters)

Every time the pages in FSM 3 send contributions into C', D', and E', this page accumulates those contributions into its variables C, D, and E (state 2). At the end of the process, the page transfers to state 3, where the area of precipitation is acquired into A (first expression); C takes the mean value of the radial (in C) and azimuthal (in E) computations (second expression); and finally, this mean and the integral echo intensity in D are divided by A (third expression). The obtained parameters, area of precipitation in A, G parameter in C, and mean value in D, are routed to the output. Then the page disappears.

6.2.1

Figure 27 represents the program in the user language of section 5.2. Such a program is also the machine program for a CPL system as implied in this report. To make the program executable by the CPL 1 machine, which has only four variables x and not all the features in the language here used, the computation has to be distributed in a larger number of states. The actual program is contained in two punch cards (Fig. 30). Results obtained with this program were shown in Fig. 22. Programs producing the same process have been written in FORTRAN and PL-1. However, a program has not yet been devised that can make standard computers do this process in real time. Fig. 28 shows the listing of one FORTRAN program, and Fig. 29 the flowchart of one PL-1 program.

In reference to the issues discussed in sections 2.1 and 2.2, we can say that Fig. 28 is a verbal representation of a process modeled in verbal form; Fig. 29 is an image representation of a process modeled in verbal form; and Fig. 27 is a symbolic representation (a mixture of words and graphics) of a process modeled in abstract machine (imagery) form.

The results of the comparison are in Table 2, expressed with the conventions described in section 6.1.3.

T A B L E 2				program 5410		
language	number of statem./ instruc.	number of symbols	number of cards	memory words	execution time per segm.	programing time
C P L 2	35	261			r t	
C P L 1	86	390	2	160	r t	
FORTTRAN	42	406	42			
P L - 1	49	273				
for definitions and criteria see section 6.1.3						

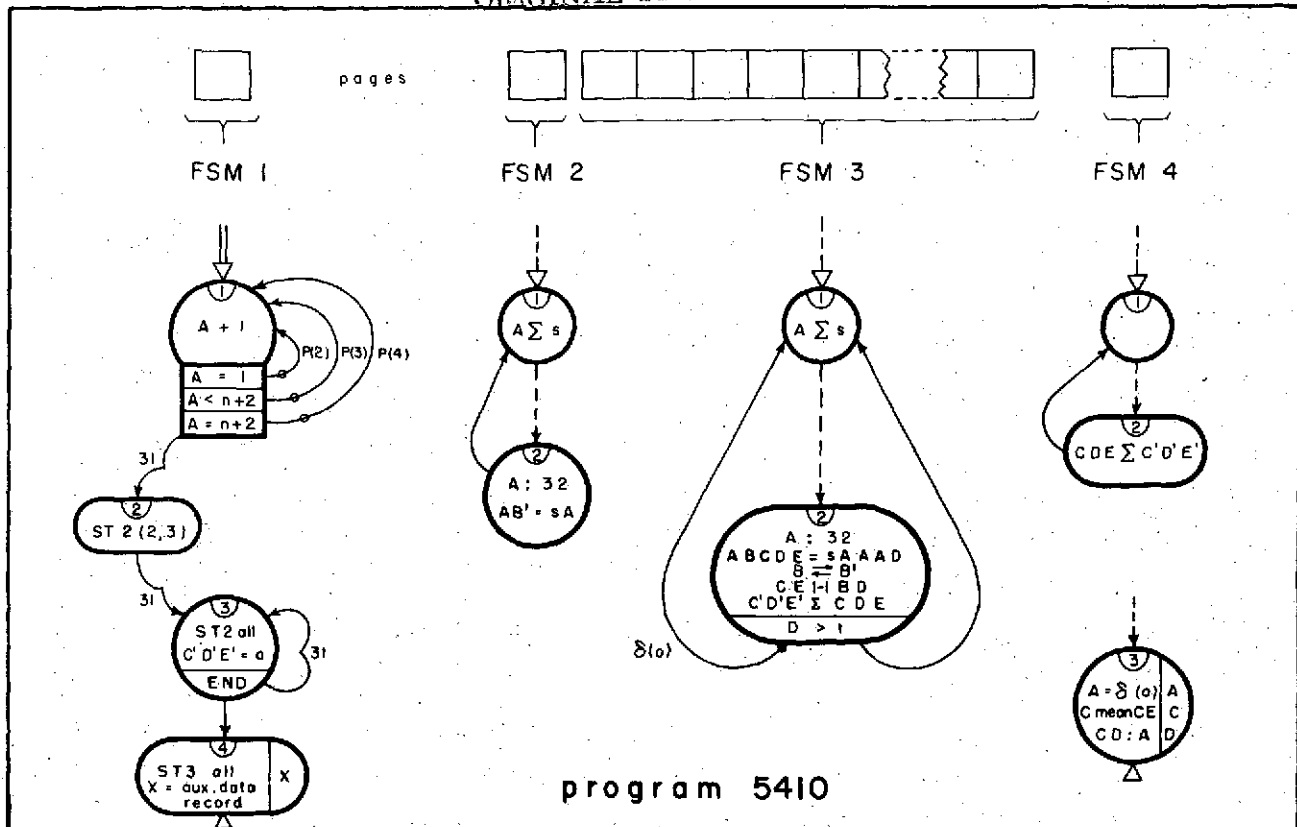


Fig. 27 - State diagram for a CPL 2 machine

```

LOGICAL K
INTEGER SUM(100), SR, ST, THRES, OLD(100)
DIMENSION NPRD(100), ISHIFT(99), IOLD(99)
EQUIVALENCE (SUM(2), ISHIFT(1)), (IOLD(2), IOL
K = .FALSE.
SR = 0
N = 0
ST = 0
45 DO 50 I = 1,100
50 SUM(I) = 0
DO 60 J = 1,28
READ (9,END=80) NPRD
DO 55 I = 1,100
55 SUM(I) = NPRD(I) + SUM(I)
60 CONTINUE
IF (K) THEN GO TO 70
SUM(1) = SUM(1)/32
IF (SUM(1).GT. THRES) THEN N=N+1
DO 65 I = 1,99
ISHIFT(I) = ISHIFT(I)/32
SR = IABS(ISHIFT(I) - SUM(1)) * SR
IOLD(I) = ISHIFT(I)
IF (ISHIFT(I) .GT. THRES ) THEN N=N+1
65 CONTINUE
OLD(1) = SUM(1)
K = .TRUE.
GO TO 45
70 SUM(1) = SUM(1)/32
IF (SUM(1).GT. THRES) THEN N=N+1
DO 75 I = 1,99
ISHIFT(I) = ISHIFT(I)/32
SR = IABS(ISHIFT(I) - SUM(1)) * SR
ST = IABS(ISHIFT(I) - IOLD(I)) * ST
IOLD(I) = ISHIFT(I)
IF (ISHIFT(I) .GT. THRES ) THEN N=N+1
75 CONTINUE
OLD(1) = SUM(1)
GO TO 45
80 RADIAL = SR/N
TANGE = ST/N
STOP
END

```

Fig. 28 - FORTRAN listing

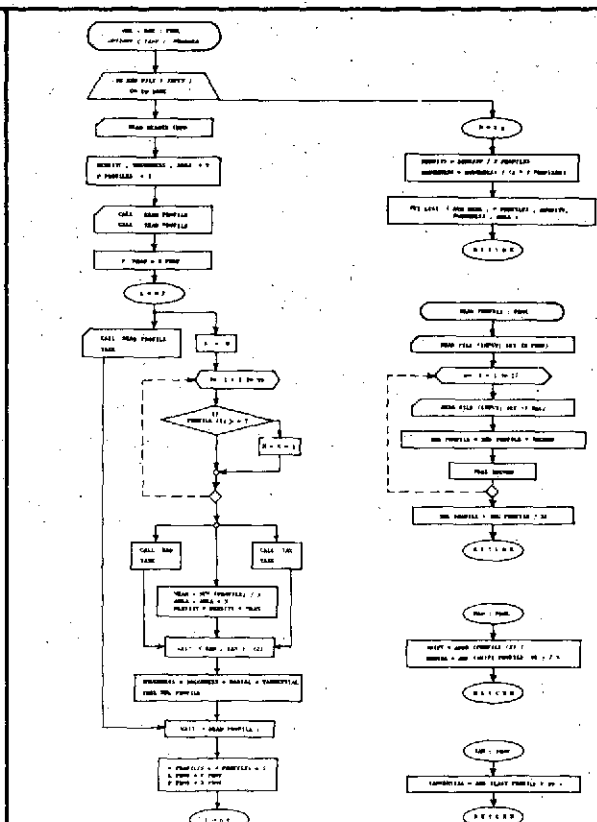


Fig. 29 - PL-1 flowchart

[illegible]

Fig. 30 - The program for the CPL 1 machine

Program 5414 in Fig. 30 produces the same process as the program in Fig. 27, except that it starts and ends at each passing of the antenna through north, for a prescribed number of times. That is, it repeats the measurements for an entire exploration of 360 degrees every other antenna rotation, say, for half an hour; and then it prints all the measurements made.

6.2.2 Distribution of precipitation intensity

In the usual model for the measurement of precipitation by means of radar, the mean echo intensity, corrected at each point to take account of several factors, is related to the rainfall intensity (Battan 1959; Austin and Geotis 1971). From an operational viewpoint, all questions on this relation that apply uniformly to all data can advantageously be postponed to the subsequent stage of data analysis; while all corrections that are specific for each point and period need to be made in real time. Among the latter, the following can be mentioned: geometrical diffusion of electromagnetic waves with distance; exclusion of nonweather echoes, such as ground echoes; attenuation encountered by the electromagnetic waves because of the precipitation itself; differences in drop-size distribution that may occur in different regions and different periods.

Experimental programs toward the above task are here described. The basic operation consists of integrating independently at each point the fluctuating radar echo, making the related corrections, and counting the occurrences of each resulting value during an entire antenna rotation. The operation should be repeated continuously, or at periodic intervals of time.

Ground echoes exhibit a much restricted spectrum of fluctuation than weather echoes do. Characteristic of weather echoes are fluctuation frequencies of the same order of radar repetition frequencies, and a constant standard deviation (Marshall and Hitschfeld 1953). Taking advantage of the above characteristics, it has been found experimentally effective to discriminate the echoes in base of the value reached by the following parameters:

$$\beta = \frac{1}{N} \sum_{n=0}^N |\log P_i(t + nT) - \log P_i(t + (n+1)T)| \quad (6.4)$$

where P_i is the echo power at point i , T the radar pulse repetition period, N a number of consecutive raw echoes, and the bars denote absolute value of the difference. Weather radars, typically, produce output signals already in logarithmic scale. Parameter β can be computed at each point in the same time in which the raw echoes are integrated for producing the echo intensity, and then compared with a threshold for deciding whether the echo should be disregarded or not. A program for echo intensity distribution, with rejection of ground echoes based on this method, is shown in Fig. 31 in different versions of the CPL language.

In the upper part of the figure, the program is shown in the language of section 5.2, and it is used for describing the process, referring to the previous example for the details. A first page performs FSM(a) that, in state 1, waits for the antenna to point to north, then configures a piece of memory into a distribution of 16 storages, and, in state 2, creates all the other pages; in state 3, it controls the synchronous behavior of the other pages, and, at the next passage of the antenna through north (in state 4) produces the output results. A sequence of pages that corresponds to the array of points in range performs FSM(b). In state 5, variable A produces the absolute value of the difference between consecutive samples, B accumulates these differences, and C accumulates the straight values of the successive echoes. After 32 such cycles, the pages transfer to state 6, where the accumulation in C is divided to form the mean value; A takes the present sample necessary for the next cycle in state 5; and transition-related routings are performed as follows. If C is below a threshold h (which means no echo is considered to be present), B and C are cleared, and no routing is made. If $C > h$, but B is below a threshold k (which means that the echo was not from a fluctuating weather target), B and C are cleared, and a 1 is added to a quantity named "rejected". If both B and C are above the thresholds, a 1 is added to a quantity named "area", C is accumulated into a quantity named T, and also distributed in the memory section created by FSM(a) at the exit from state 1.

In the middle part of Fig. 31, the program is adapted to the CPL 1 machine; some states are added, and some different allocations of variables are made. The starting of the FSMs from states 0 and 8 is for using an existing supervisor that initiates FSMs in those states every minute. The lower part of Fig. 31 shows the program in machine language; all the items constituting the state diagram can fit into one punch card. Comparison with equivalent programs written in various programming languages gives the data summarized in Table 3.

In this program no account was taken of geometrical considerations. If a large span of range is involved, each point should be weighed proportionally to the range, because an element in polar coordinates has an area that increases with the radius. This can be accomplished by multiplying the measurement at a range R_1 by the factor R_1/R_0 , being R_0 the range at which the factor is assumed to be equal one. This in the hypothesis of antenna beam always filled.

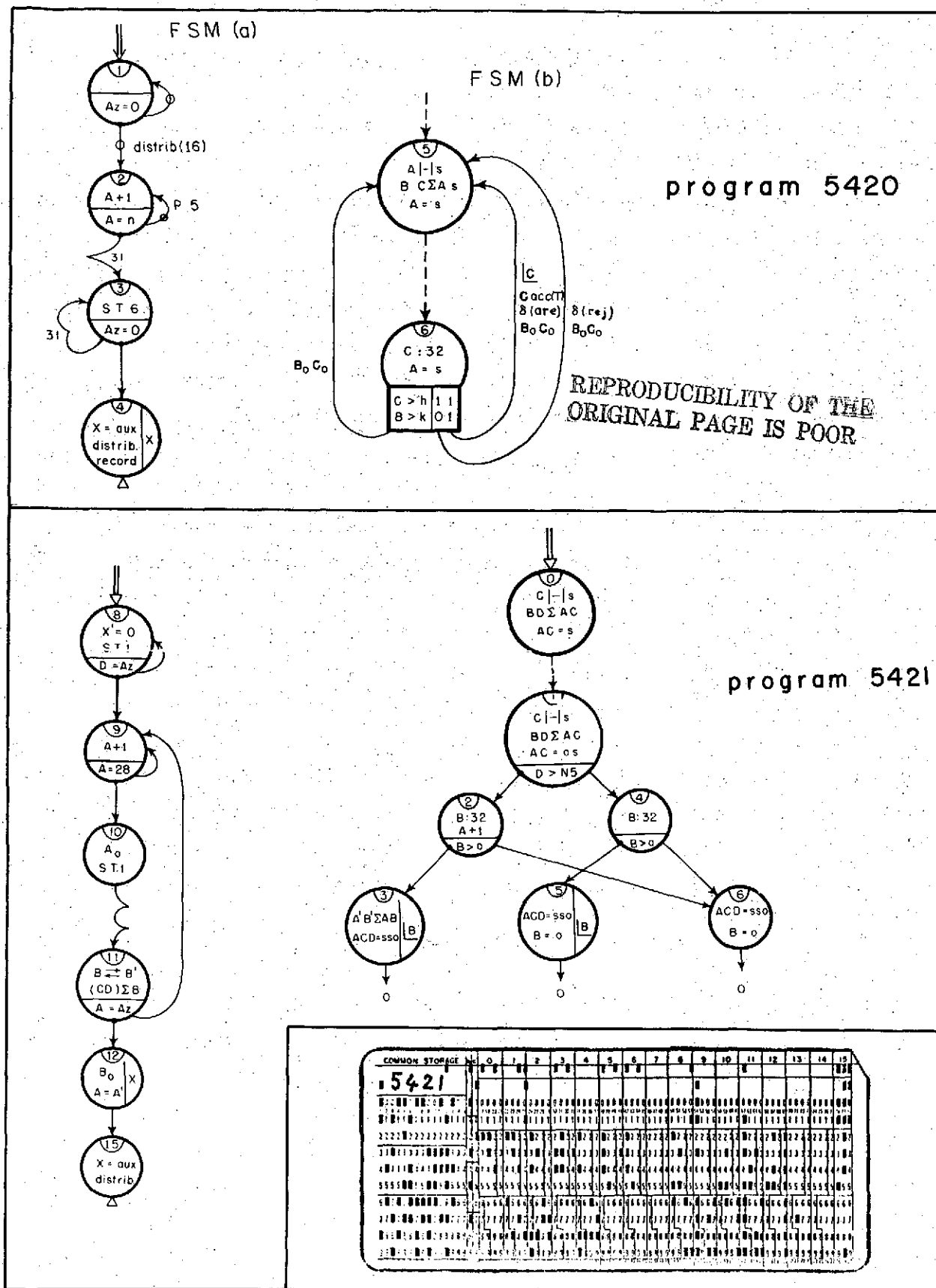


Fig. 31 - Three representations of the same process

T A B L E 3				program 5420 / 1		
language	number of statem./ instruc.	number of symbols	number of cards	memory words	execution time per segm.	programing time
C P L 2	26	122	1	74	r t	
C P L 1	40	193	1		r t	
for definitions and criteria see section 6.1.3						

The correction because of the geometrical diffusion of electromagnetic waves is usually made by adding a coefficient that increases with range to logarithmic signals. More interesting is the correction of the attenuation due to the precipitation itself. This is an integral function of the rain intensity encountered along the range. But the rain intensity can be related to the reflectivity factor Z measured by the radar; therefore, the specific attenuation γ can be expressed in the form $\gamma = k Z^h$. In the case of logarithmic signals, a discrete algorithm for the attenuation correction can be as follows:

$$X_r = m_r + C'_r$$

$$C'_{r+1} = C'_r + qX_r$$

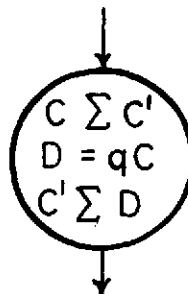


Fig. 32

Computation of the
attenuation correction

where m_r is the measurement at range r , C' an auxiliary variable for the attenuation correction, q a constant, and X_r the corrected reflectivity factor. This algorithm can be implemented by an additional state (Fig. 32) to be inserted in FSM b of the previous program (Fig. 31). Variable C in the consecutive pages computes first m and then X ; the auxiliary variable C' computes the term C' in the above expression.

6.2.3 A compounded program

To give an example of the flexibility that is available when structuring the programs in FSM form, we show how to compose a new program from parts of others.

Program 5410 (section 6.2.1) produces the interesting parameter G that can be well related to the type of precipitation. But in fact, we are most interested in knowing when and where G exceeds a certain value. Program 5420 (section 6.2.2) discriminates the nonweather echoes; we would like to have this feature also in the measurement of G. The intensity distribution of program 5420 is also information that we would like always available.

To implement the above desires, we compose program 5430 represented in Fig. 33. An FSM 3 has states 1 and 2 analogous to those of FSM(b) of program 5420 for measuring the mean value of the echoes and discriminating the nonweather echoes. Then state 3, similar to state 2 of FSM 3 in program 5410, is added for measuring the G parameter. An increment of one in A' is added in this state, for computing a local precipitation area. A further state 4 is necessary for the case of nonweather echo.

FSM 4 has states 1 and 2 analogous to states 1 and 2 of FSM 4 in program 5410. But here a local G is computed immediately in state 3, and if it exceeds a value m, the azimuth, elevation, and range are read (state 4) and all the data are sent to the output.

FSM 2 is similar to FSM 2 in program 5410, except that it includes the algorithm for nonweather echo rejection. The control is performed by FSM 1, which is a compound of the FSMs 1 and (a) in programs 5410 and 5420, respectively.

This program has been adapted to the CPL 1 machine, and its coded form is shown in the two-punch cards of Fig. 34.

A real-time printout obtained with this program is shown in Fig. 35. Each line above the flag 7777 indicates a region where G was larger than a given value; the four numbers in these lines are from left to right: azimuth, area, mean intensity, and G. The line above and one below the flag contain auxiliary data. The last four lines contain the sixteen values of the intensity distribution.

The comparison with programs written in different languages is in Table 4.

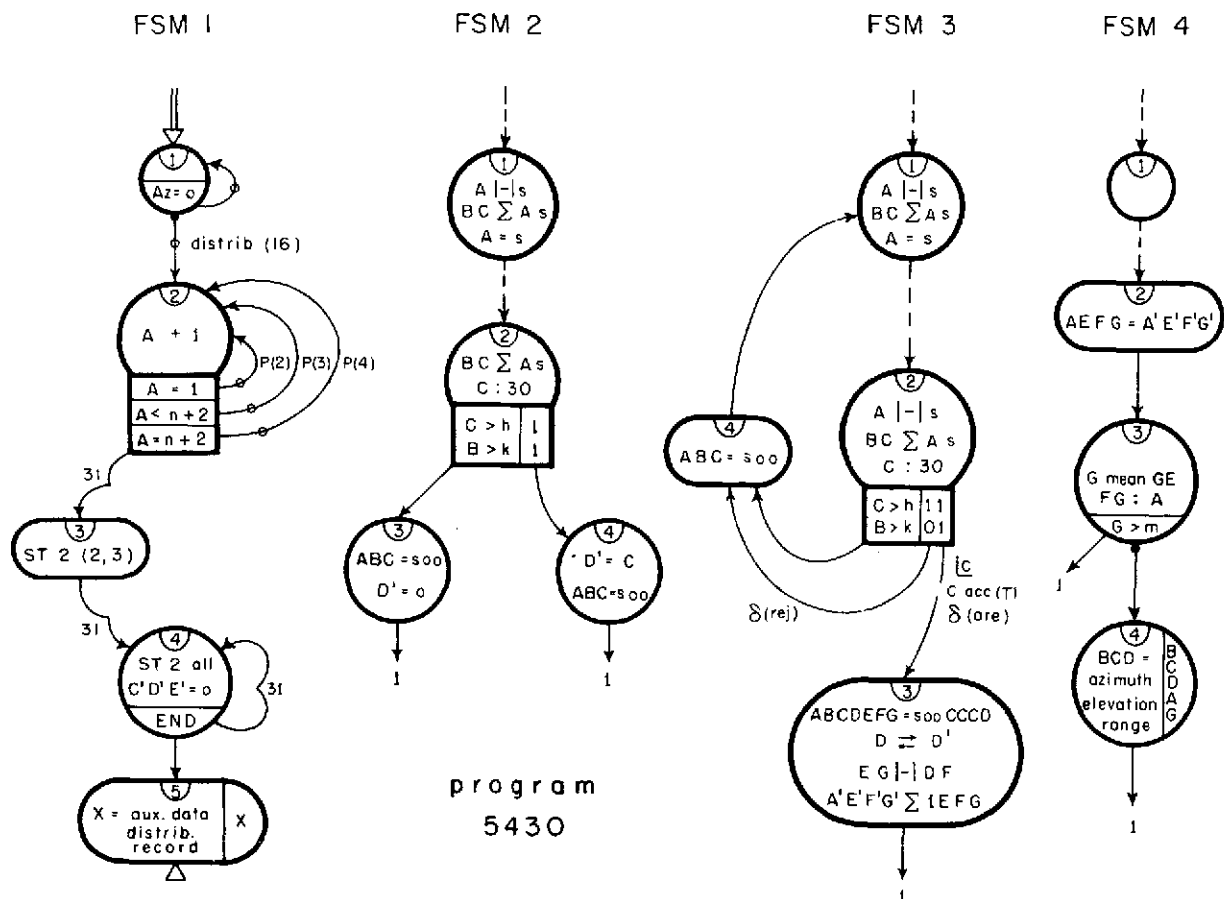


Fig. 33 - State diagram compounded from different programs

T A B L E 4				program 5430 / 1		
language	number of statem./ instruc.	number of symbols	number of cards	memory words	execution time per segm.	programing time
C P L 2	64	322	2	142	r t	
C P L 1	75	353			r t	
for definitions and criteria see section 6.1.3						

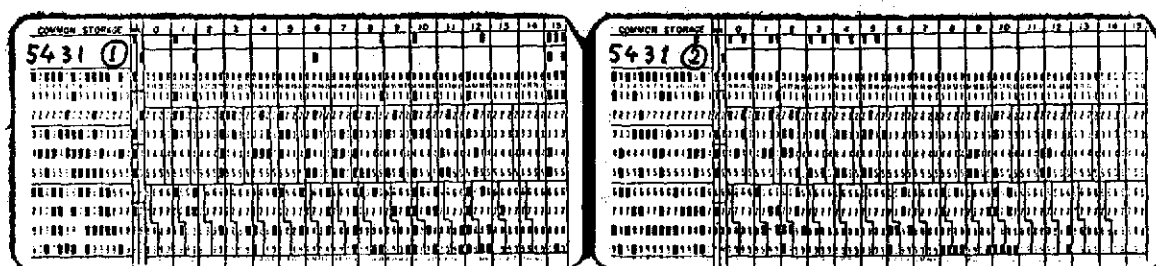


Fig. 34 - The CPL 1 machine program

0252	0004	0004	0004
0252	0009	0009	0009
0248	0009	0040	0009
0244	0009	0046	0009
0244	0004	0030	0044
0242	0009	0021	0009
0238	0004	0050	0009
0234	0004	0030	0009
0228	0010	0036	0044
0226	0009	0029	0009
0224	0012	0055	0030
0224	0010	0035	0030
0222	0012	0040	0041
0220	0008	0043	0002
0212	0008	0044	0009
0210	0008	0020	0002
0166	0011	0040	0041
0166	0011	0053	0030
0035	0008	0045	0009
0032	0003	0008	0009
0032	0005	0022	0044
0030	0012	0024	0002
0030	0010	0021	0002
0028	0010	0020	0009
0024	0012	0022	0040
0020	0000	0000	0040
0002	2004	1111	0140
0012	0000	0310	2240
3340	1544	0111	0424
0218	0128	0100	0061
0035	0001	0014	0001
0003	0004	0000	0000

Fig. 35 - Real-time printout from program 5431

6.2.4 Recording of weather echoes

This example is interesting for showing the flexibility that is available for interfacing with the environment.

We want the radar echo measured with an integration of 64 samples, with exclusion of ground echoes, for the entire range, every degree of the antenna rotation, regardless of its velocity. The measurements should form a record every degree, with azimuth, elevation, and initial range first, and then the echo at the successive range points, packed in two measurements for each 12-bit word.

The structure of the program, as usual for this type of processing, consists of an FSM(a) for the control, and an FSM(b), implemented by a page per range point, for the measurements (Fig. 36).

FSM(a), in state 1, reads the present azimuth, increments it by h (the equivalent of one degree), then waits idle for 63 circulations and goes to state 2. Here a driven transition to state 6 is produced, and a record is activated with azimuth, elevation, and range as first data. After two idle circulations, in state 3, a driven transition holds the other pages in state 4 until the present azimuth has increased by one degree; then, the same routine repeats until a stop signal appears (transition in state 2).

FSM(b), in state 5, performs the algorithm described in the program of section 6.2.2 for measuring the mean echo value and at the same time providing a parameter for discriminating the ground echoes. In state 6, this parameter is tested against a threshold t , and, if it is smaller than t , the measurement in C is cleared by routing. With the algorithm of state 7, the pages transfer every other to state 8 or 9. For the pages which transfer to state 8, the measurement is shifted half a word to the left and sent to the auxiliary variable C' . For the pages which transfer to state 9, the content in C' is added to the present content in C and routed to the output, thus implementing the packing of two measurements in one word.

A family of programs of this type has been used for a variety of experiments. An example was shown in Fig. 24. For the CPL machine, these programs are contained in one punch card. The comparison with equivalent programs in other languages is in Table 5.

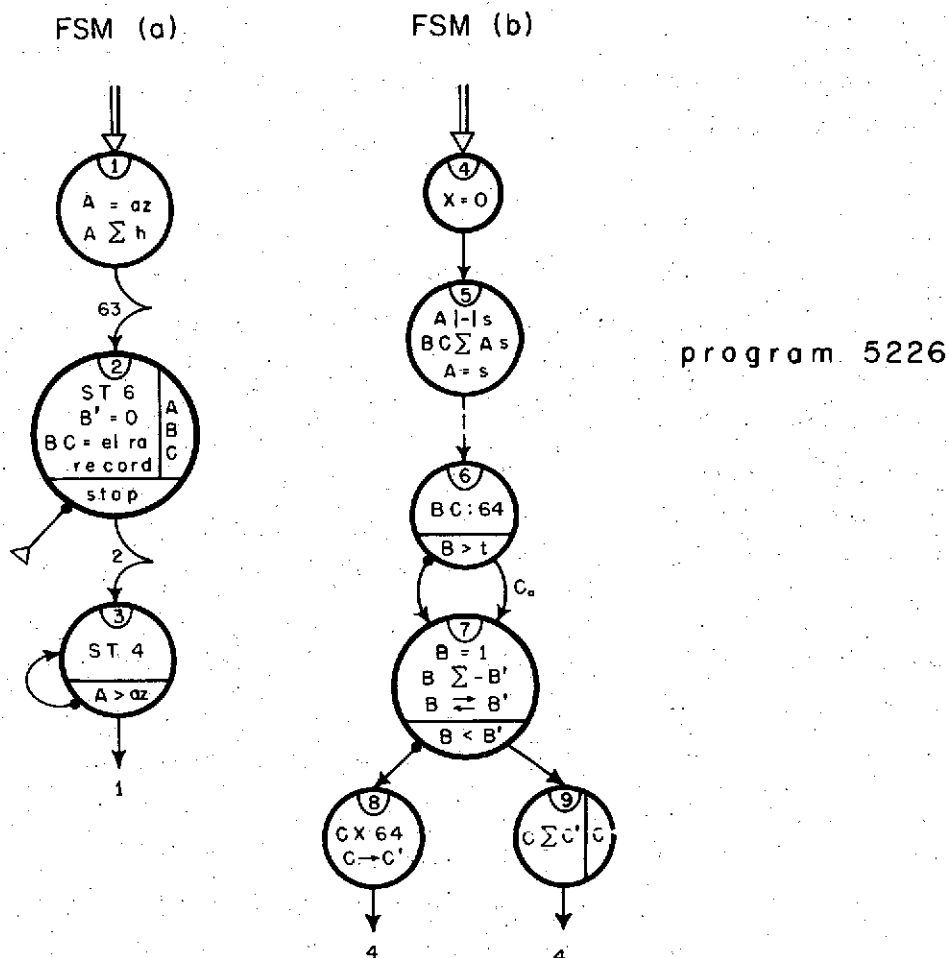


Fig. 36 - State diagram of the program for the CPL 1 machine

T A B L E 5				program 5226		
language	number of statem./instruc.	number of symbols	number of cards	memory words	execution time per segm.	programing time
C P L 1	27	120	1	68	r t	
for definitions and criteria see section 6.1.3						

6.2.5 Measurement of statistical characteristics of weather echoes

A program is here shown as an example of the variety of processes that can easily be conceived for analyzing the characteristics of weather echoes (see also Schaffner 1972 c).

We saw in the last three examples the usefulness of parameter β . There is an interest in its relation to the rate of crossing the mean echo value, of which there are analyses available (Fleisher 1953). Also, its possible dependency on the echo intensity needs to be explored.

For this purpose we may wish, for all the points of an entire region of precipitation, the echo intensity, the β value, and the crossing rate. In order to measure the rate of crossing of the mean value, the latter has to be measured first; to be aware of possible changes that may occur in the echo intensity, we measure it a second time, at the end of the other measurements. Such measurements are provided in the following program.

After having described the previous programs in the user language of section 5.2, this program will be described directly in the restricted language of the CPL 1 machine. As usual, there is an FSM(a) for control (Fig. 37), and an FSM(b) implemented by a sequence of pages for the measurements.

The control FSM determines four phases of processing by means of driven transition to states 1, 6, and 7, in states 10, 11, and 12, respectively. The duration of these phases is determined in state 13 by the overflow of D which accumulates the content that has been previously set in B.

In the first phase, all the measuring pages accumulate 128 video samples sam (state 0); this accumulation, after division 128 (state 1), constitutes the preliminary mean echo value, located in variable A, against which the crossing will be tested.

In the second phase, the pages run through states 2, 3, 4, and 5 for measuring the parameter β and for counting the crossings. In each of these states, the difference is made in B between the previous sample and the present sample; this difference is accumulated in D; and then the present sample is stored in B for the next cycle. Moreover, following the tests in these states, we can observe that if the sample has been previously less than A (the reference mean), the page stays in state 2; if the sample has been previously larger than A, the page stays in state 3; but every time the present sample crosses the value of A, the page goes for one cycle either to state 4 or to state 5, where, furthermore, a one is added into variable C. This

phase lasts for 512 radar periods. In this FSM we see an example of the use of the states for memorizing past events.

In the third phase, 128 video samples are accumulated in B (state 6) in order to produce the post mean. In the last phase, the four measurements are packed into two words and routed to the output (state 8).

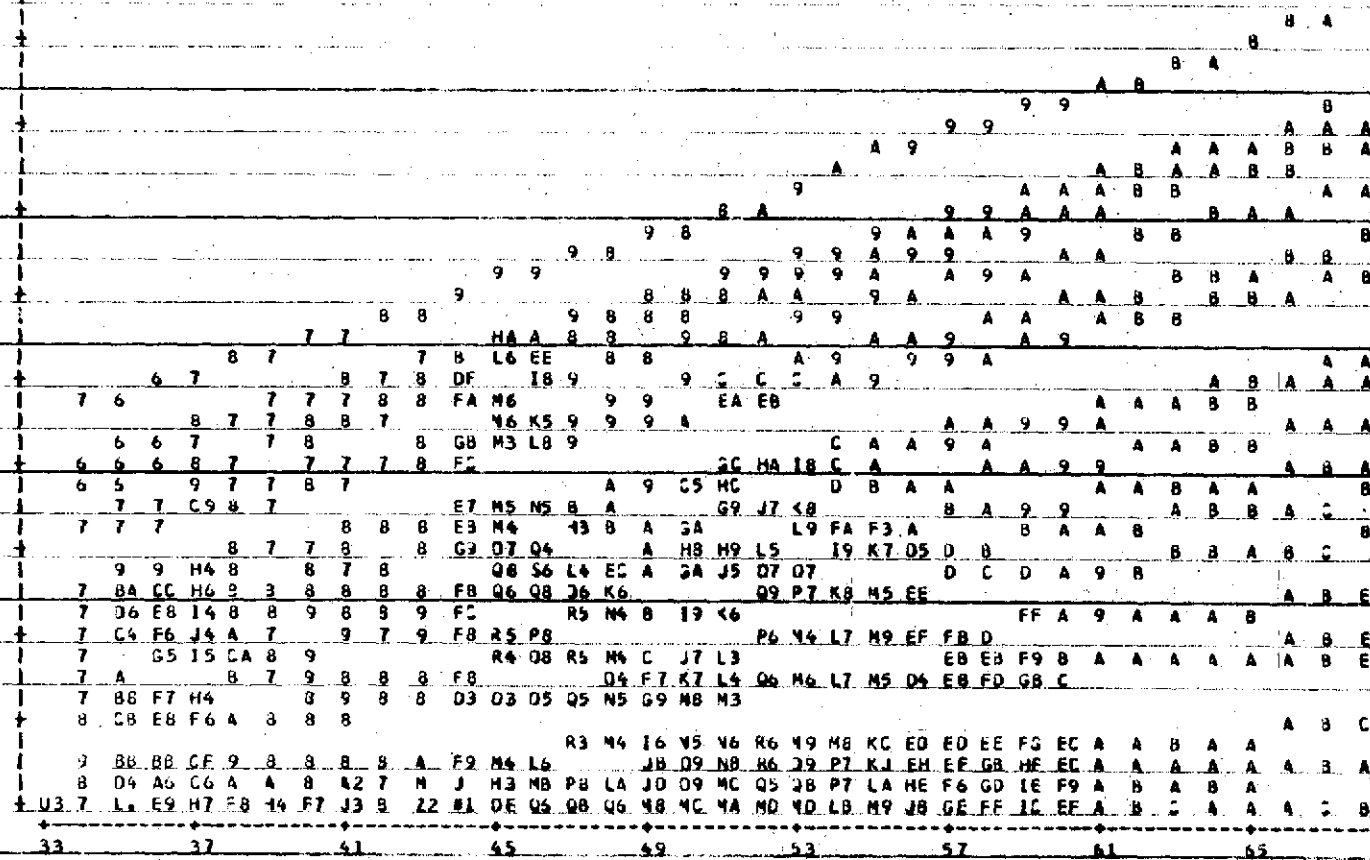
This program, when expressed in the codes for the CPL 1 machine, is contained in one punch card, as shown on the lower part of Fig. 37. The state fields in the card bear the same label as in the state diagram, and the items in the card can be easily interpreted in terms of the items in the state diagram (see section 5.3).

The comparison with equivalent programs written in different languages is in Table 6.

The crossing rate can be related to turbulence in the atmosphere. Thus, this or similar programs can be used also for data collection. Fig. 38 shows a printout of an RHI (range altitude representation) across a storm on August 14, 1972. For each point there are two values (expressed with numbers and letters in their normal sequence); the left value indicates the echo intensity, and the right value indicates the crossing rate. When the echo intensity is below a certain value, the crossing rate is not shown because it pertains mainly to the noise fluctuations.

T A B L E 6						
program 5241						
language	number of statem./ instruc.	number of symbols	number of cards	memory words	execution time per segm.	programing time
C P L 1	47	181	1	80	r t	
for definitions and criteria see section 6.1.3						

HEIGHT (KM) RHI 21 SEC 34 MIN 16 S 16 D 8 MON 1972 AZIMUTH 240 PROGRAM 5232 (REFACROSSING) PRINTED 72/08/23



REPRODUCIBILITY OF THE
ORIGINAL PAGE IS POOR

Fig. 38 - Echo intensity and mean crossing from program 5232

6.2.6 Measurement of the dispersion of short means

An experimental determination of the dispersion of echo-intensity measurements obtained by averaging a few samples is interesting in two ways. First, it gives a knowledge of the actual confidence that can be assigned to the intensity measurements for different types of meteorological events. Second, it gives information about the decorrelation time of the echoes at a point; information that can be related to the movement of the scatterers in the atmosphere. A practical organization of this type of measurement is described below.

We first want a precise measurement of the actual mean value, say by using 1000 samples. Then we want several measurements (say m measurements) with a few samples (say n samples). Furthermore, we would like these few samples taken at different time distances; say, k radar periods are skipped between samples. Finally, because the actual intensity might have varied during the period of the measurements, we want another precise measurement made again with 1000 samples. We would like all the data printed out, with the short measurements already arranged in a distribution.

The above is a conceptually very simple process. But it is of the type whose procedural description is awkward when recited in sentences. An abstract machine can represent more concisely the entire task. If, moreover, this abstract machine can be used directly as an actual program, the entire work of the experimenter is simple indeed.

An abstract machine that naturally derives from the above description is represented in Fig. 39. It is one FSM performed by one page; the samples s , and the parameters m , n , and k are treated as input data. In state 1, variable A accumulates the samples, and the page "stays" 1000 times in this state. Then A is divided by 1000 and routed to the output (state 2); this is the first precise mean. At the exit of this state, a reservation for a 32-cell distribution is routed. Then, in state 3, variable A accumulates samples, while B increments by one. If no radar periods should be skipped ($k = 0$), the page remains in state 3; otherwise it goes to state 4 for the number of circulations to be skipped as indicated by k . Every time the samples accumulated are n , the page transfers to state 5, where the mean value is obtained by division and routed to the distribution function. Variable D is incremented by one, and then the measurement is repeated, returning to state 3.

When m measurements are executed (test in state 5), the page stops over state 1 again (the "stay" prescription holds the page in state 1 for 1000 circulations), then stops over state 2, where the new precise mean is routed to the output, and finally transfers to state 6. Here the parameters n , m , k , and possibly other auxiliary data are read and routed to the output; the content of the distribution is added to the output buffer; an output record is ordered; and then the page disappears.

This program, when adapted to the CPL 1 machine, has minor variations, and it is still contained on one punch card. The comparison with equivalent programs in other languages is in Table 7.

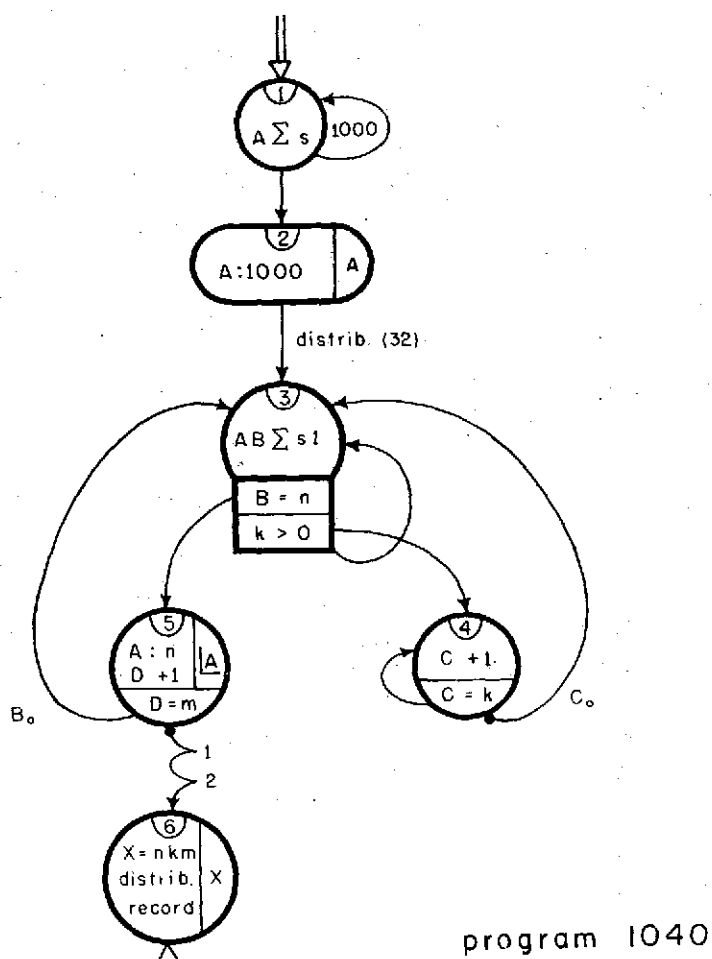


Fig. 39 - Program for the dispersion of short means

T A B L E 7				program 1040 / 1		
language	number of statem./instruc.	number of symbols	number of cards	memory words	execution time per segm.	programing time
C P L 2	23	102	1		r t	
C P L 1	28	130	1	52	r t	
FORTTRAN	36	302	36			

for definitions and criteria see section 6.1.3

It is interesting to note that the high-level-language FORTRAN program is less concise and less problem oriented than the machine-language CPL program. Fig. 39b shows the connections among routines of the FORTRAN program.

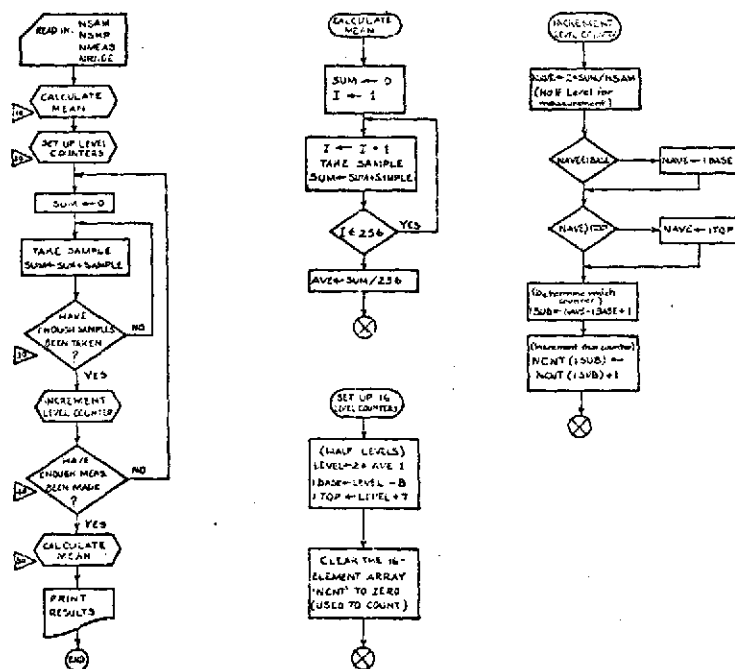


Fig. 39 b - Flowchart of the corresponding FORTRAN program

6.2.7 Real time data handling

Certain types of processes are so related to environmental contingencies that high-level programming languages are hardly applicable, and programs are in fact prepared in machine language. In the case of the CPL system, there is no real difference between the user and the machine languages, in the sense that the user language allows complete control of the execution, and the machine language has almost the conciseness of a user language. A simple case is here described.

A measure of interest is in comparing precipitation intensities deduced from radar data with actual precipitations measured at the ground. For this purpose we want measurements of the radar echo intensity at an array of 5×5 points, 1 mile apart, in a region near Concord, Massachusetts, where a network of rain gauges is located. The measurements should occur automatically during the rotation of the radar antenna, and a printout with related auxiliary data should be produced at each passage of the antenna through the Concord region.

Given the small area, azimuth and range can be taken with good approximation as coordinates of the grid points. At each point, a number n of consecutive echoes is integrated. These results are multiplied by a constant and another constant is subtracted, in order to express the measurements in the desired unit. Then these numbers are printed in the form of a 5×5 matrix, along with the date, time, number of samples, and coefficients used. This program has been written in machine language for two systems: the CPL 1 machine directly connected to the environment, and for a system composed of a PDP 8/1 minicomputer and special units for the connection with the environment.

The state diagram of the CPL program is given in Fig. 40. One page performs the control FSM (a), and five pages located at the proper range segment perform FSM (b). At the initial azimuth a_0 (test in state 8), a driven transition brings the five pages to state 1, and then to state 2, where the accumulation of the samples is made. After 100 circulations, a driven transition brings the pages to state 4, where the coefficient n_1 is subtracted and the result transferred to variable B; then the division by n_2 is performed in state 6. If at the start of the new measurement the division is not finished, it continues in state 3, together with the accumulation of new samples in A.

After the first measurement, the transfer to state 4 is substituted by the transfer to state 5, where the measurement obtained in C is routed to the output. After five measurements (test in state 11), the parameters n_1 , n_2 , and elevation are sent to the output and the printout is started. The program is contained in one punch card. Ninety 12-bit words are needed in the memory, including supervisory instructions.

Of the program for the PDP 8/1, the connection of the routines is shown in the flowchart of Fig. 41. The routines are not described here. The total program occupies 2 K of 12-bit words. Data of comparison between the two programs are given in Table 8.

T A B L E 8						
program 1202						
language	number of statem./ instruc.	number of symbols	number of cards	memory words	execution time per segm.	programing time
C P L 2	36	144	1			
C P L 1	38		1	90	r t	
FORTTRAN						
machine lan. (PDP 8/1)	840		500	2000	q r t	
for definitions and criteria see section 6.1.3						

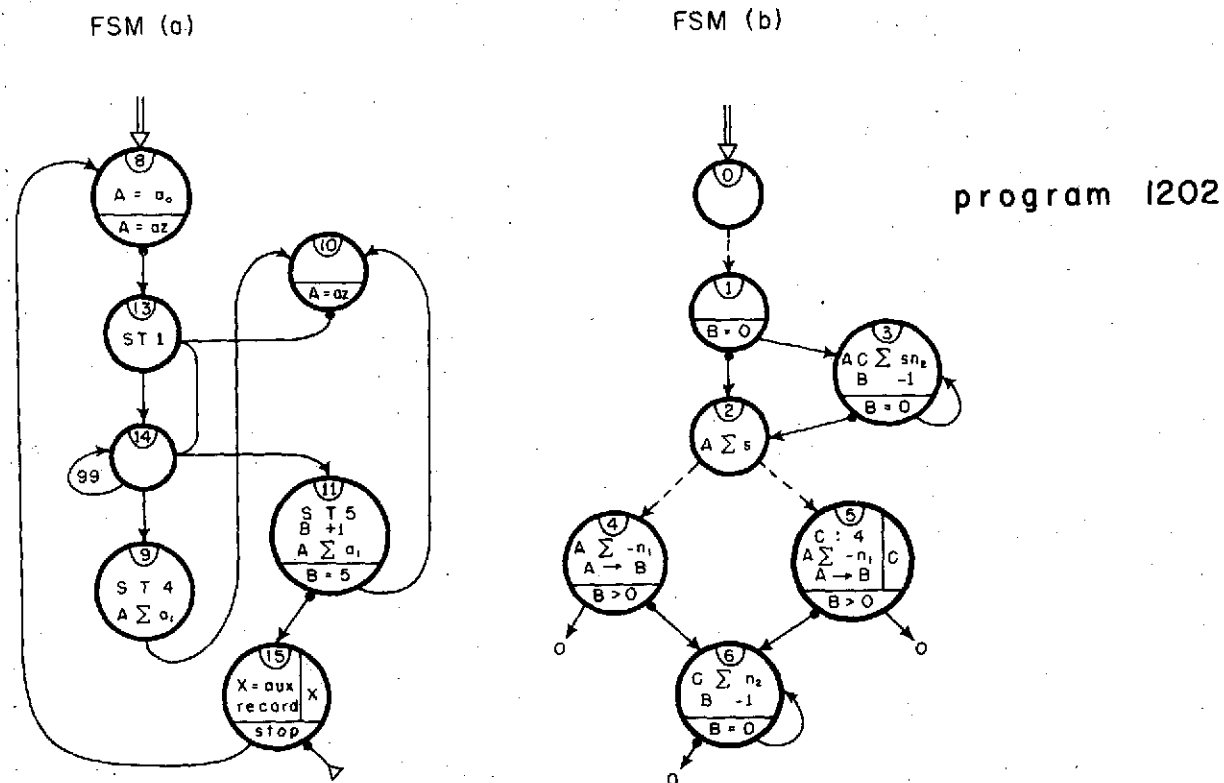


Fig. 40 - The complete program for the CPL 1 machine

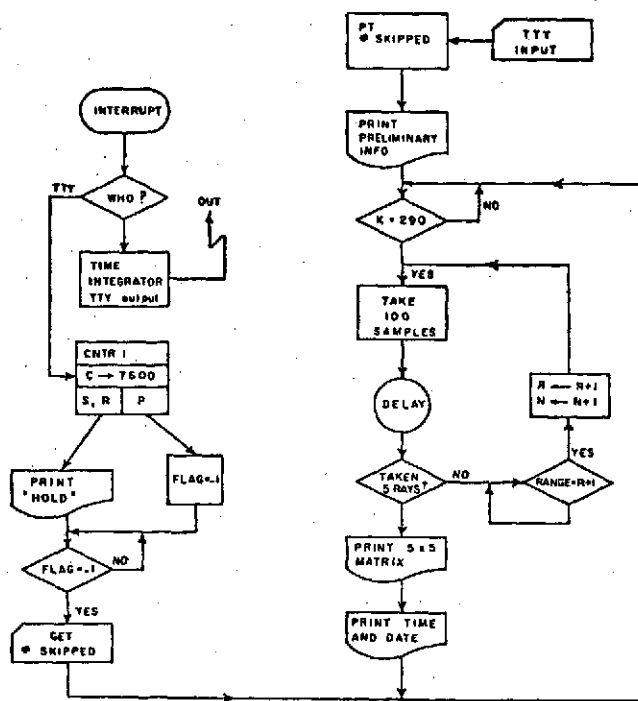


Fig. 41 - The flow between routines in the conventional program

6.2.8 Measurement of the antenna pattern with solar noise

As mentioned at the beginning of section 6.2, the solar noise offers an interesting solution to the need of antenna calibration. Among the advantages of the sun as a source are optimum position in the sky, regular and known movement, absence of frequency and interferences problems, no cost. Among the disadvantages are very low signal intensity, fluctuations of different types (e.g., solar flares), angular extension (about 0.5 degree). With sufficient integration, the weakness of the signal ceases to be a problem. By repeating the measurements, the momentary fluctuations can be recognized. The effect of the extension of the emitting region of the sun can be separated by reverse convolution with the profile of the noise intensity that is normally published for each day. The gain of the antenna can be computed from the geometry of the antenna pattern, regardless of the absolute value of the received signal. The system response to the small signal variations is deduced by using the same procedure for a source of noise with calibrated attenuator.

The program here described produces a two-dimensional antenna pattern by giving the antenna a small vertical scanning across the horizontal trajectory of the sun at around noon. Numbers n_1 and n_2 , indicating the lower and upper elevations of the segment of interest in the antenna scanning, are established and set as input data. The total number of points desired is set as input datum n_3 . An input value n_4 is experimentally chosen for properly positioning the zero in the present noise intensity. The entire radar range is filled with pages implementing FSM (b), Fig. 42, and one page implements FSM (a) for controlling the measurement.

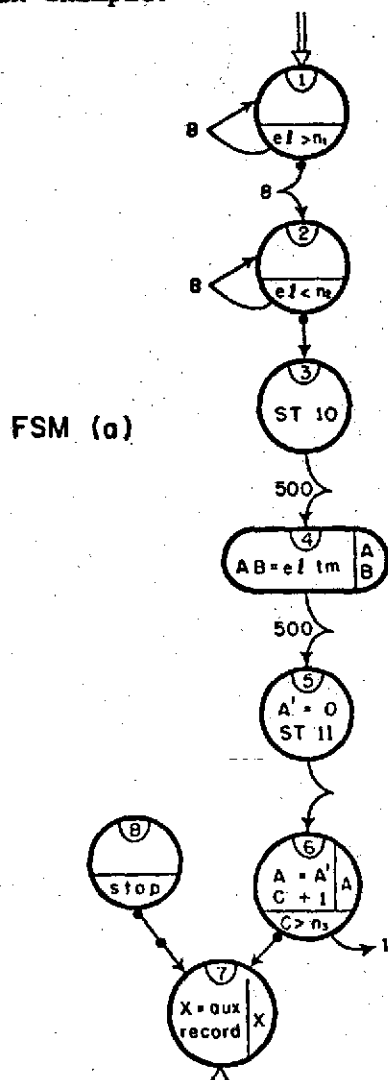
The measuring pages, in state 10, accumulate in A all the noise samples that can be considered independent in the range interval occupied by the page, say four samples. This accumulation is made with free overflow, thus the number of samples that can be accumulated is unlimited. In state 11, the value n_4 is added, and the result is accumulated into the corresponding variable A' in Ω' . Variable A is cleared, and the pages go to the waiting state 9.

The control page waits in state 1 or state 2 until the antenna is in the prescribed interval of elevation. Then state 3 produces a driven transition to state 10. In state 4, which occurs at midpoint of the integration in the measuring pages, the present elevation and time are routed to the output.

In state 5, A' is cleared, and a driven transition to state 11 is produced. In state 6, the present content of A' is acquired and routed to the output, variable C is incremented by one, and the routine repeats itself.

When the antenna is outside the prescribed interval of elevation, the control page rests in either state 1 or 2, and the measuring pages in state 9. When the prescribed number of measurements is reached, transition to state 7 occurs, auxiliary data are routed, and a record is produced. If a record is desired at any anticipated moment, a signal STOP is activated, and when the control page stops over state 8, transfer to state 7 occurs immediately, with the consequent production of a record.

The output consists of triplets: an elevation, a time (from which an azimuth can be derived, given the known movement of the sun), and an intensity. From these triplets the antenna pattern can be constructed. Fig. 25 shows an example.



program 1710

FSM (b)

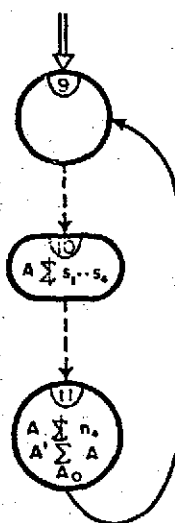


Fig. 42 -

Program for antenna pattern measurement

Comparison with equivalent programs written in different programming languages is in Table 9.

T A B L E 9				program 1710		
language	number of statem./ instruc.	number of symbols	number of cards	memory words	execution time per segm.	programing time
C P L 2	25	112	1	76		
C P L 1	31	128	1			
for definitions and criteria see section 6.1.3						

6.2.9 R. A. D. A. R.

This acronym stands for Radar Data Acquisition and Registration system, and refers to a software system under preparation at the Politecnico di Milano, Italy, for a weather radar to be used in connection with the satellite Sirio experiments (Cantarella et al 1971). The task of this system is briefly described, and then an implementation for a CPL system is shown.

In these experiments, the radar scans the volume of the atmosphere involved in the path between the satellite and a ground station. At each revolution of the radar antenna, a frame of data should be analyzed, and if at some points of the frame a weather target is found, the frame should be recorded on a magnetic tape. A frame of data is constituted of adjacent "rows" between two assigned azimuths a_1 and a_2 . A "row" is constituted by 1024 adjacent range points, 75 m apart, in each one of which 128 consecutive echoes are integrated and corrected with a range normalization coefficient rn . If in any row of a frame at least h points show intensity above a threshold t , that frame should be recorded. If echoes continue to be present, the recording of all the frames should continue; if the echoes cease, the recording should stop after one further frame without echoes. If echoes appear again, the recording should resume. At the end of each frame (also if no measurements are recorded), a record should be produced with auxiliary data such as date, time, elevation, and messages. After m frames, regardless of whether echoes were present, the process should end.

An implementation of this process with a CPL machine is shown in the state diagram of Fig. 43. An FSM (a) is implemented by one control page, and an FSM (b) by the 1024 pages at the range points. In FSM (b), the pages flow through the loop of states 10, 11 (or 12), 13, 14, (15), and again 10. In state 13, the accumulation of the samples occurs, and in state 14 the mean value is obtained and normalized. If a page produces a measurement above the threshold t , it transits through state 15, where variable A' in Ω' is incremented by one. When recording is underway, all pages transit through state 12, where the measurement is routed to the output.

The control page, in state 1, produces the sequence of measuring pages, in the usual manner, and in state 2 waits for the initial azimuth before transferring to state 3. In state 3, the auxiliary data are routed to the output; then the loop of states 4, 5 and 6 follows. The driven transition

to state 11 brings the other pages to initiate the integration, and the driven transition to state 14, after 128 circulations, brings the pages to complete the measurements. If more than h points with echo are found in A' (state 6), the control page follows the loop of states 5, 6, and 7. In state 7, the auxiliary variable B' is set to the value 2. As a consequence of being $B' > 0$, the control page will transit through states 7 or 8 for the remainder of the frame; this means that all the measuring pages will route to the output (because of the driven transition to state 12). At the end of the frame (azimuth larger than a_2 , tested in state 5), a record is produced, and B' becomes one (state 9). In this condition, the following frame will still be recorded also if no echoes are present. After one frame without echoes, B' becomes zero, and the recording ceases. At each frame, variable C is incremented by one (state 9); and when C is equal to m , the process ends, after the production of the last record (state 9).

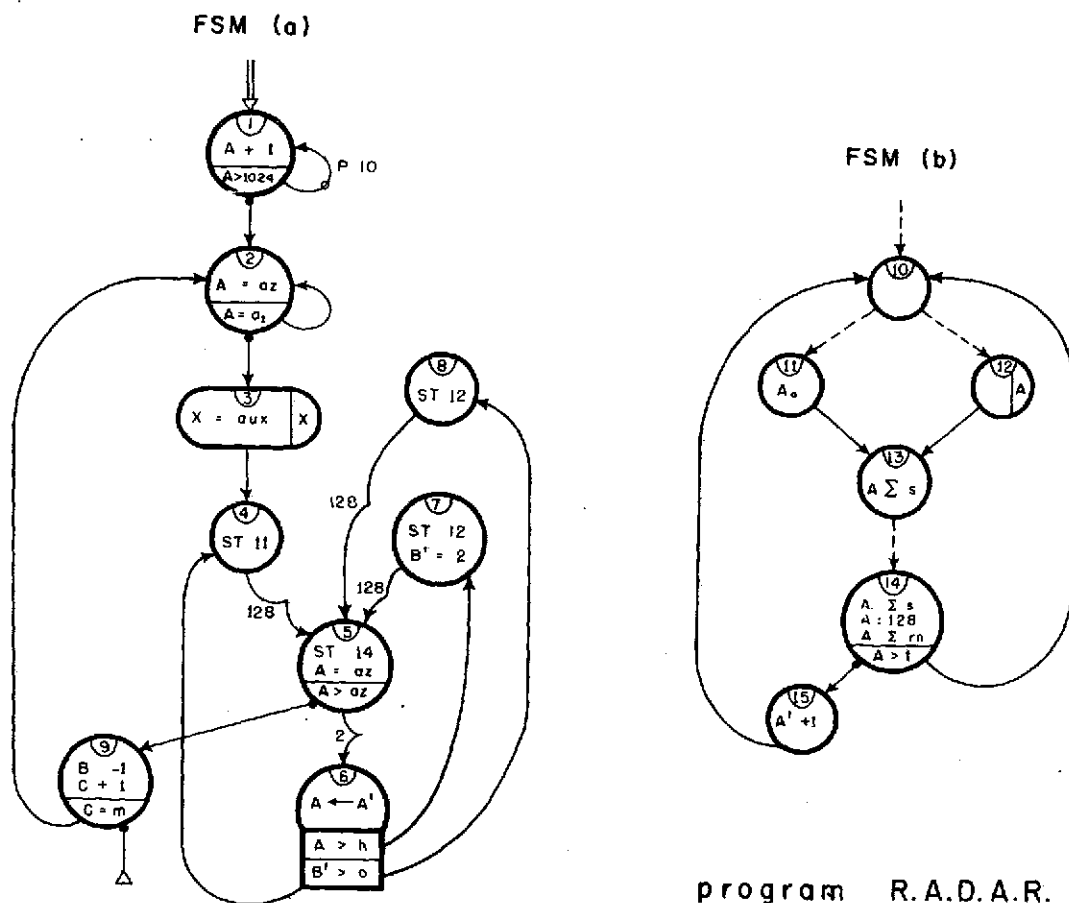


Fig. 43 - Program for automatic data collection

The data of this program that are suitable for comparison of size and complexity are reported in Table 10.

T A B L E 10 program R.A.D.A.R.						
language	number of statem./ instruc.	number of symbols	number of cards	memory words	execution time per segm.	programing time
C P L 2	36	113	1		r t	
for definitions and criteria see section 6.1.3						

6.2.10 Real-time numerical models

An observer of weather radar displays is naturally led to perform two types of mental process: (1) to interpret the patterns visible in the displays in terms of what might be the invisible (for the observer) physical patterns in the atmosphere; and (2) to anticipate mentally the future evolution of those patterns in terms of the patterns visible on the displays.

In the interest of effectiveness and precision, we attempt here to parallel these mental processes with computer processes. Some examples of processes of type (1) can be seen in the programs previously described. Now we show a preliminary example of type (2). The interest for a short-term forecast of radar echo patterns is both for predicting the area where precipitation will occur, and for anticipating the possible occurrence of severe events.

Note that the real interest is in the patterns in the atmosphere, but, because the observer (radar plus human) has access only to the radar echo patterns, any model on the evolution of the atmospheric patterns has to be based on radar data, in addition to general information available from other sources. In this situation one might be involved not in modeling the phenomena that occur in the atmosphere, but in searching for semi-empirical models of the evolution of the echo patterns per se. To give simple examples: if there is a constant wind, the echo patterns will have a linear translation; if there is also growth and dissipation, the echo patterns will have an apparent movement that is the result of several effects; if there is a continuous supply of moisture, scattered cells will diffuse in one solid pattern; if dry air is succeeding, the present pattern will dissipate with a certain time behavior.

Clearly, there are two phases: (a) the establishment of laws of time evolution for the patterns; (b) the establishment of the conditions that select one or more of these evolutions. Phase (b) can be tackled in terms of processings of type (1) above, in terms of processings for determining the past movement and evolution of the echo patterns (Schaffner 1972d), and in terms of general information available from other sources. Phase (a) can be approached with numerical models of the type used for the dynamics of fluids. A preliminary example of such models is described in this section.

In the context of the evolution of weather-radar echo patterns, we call

x_i the echo intensity at point i in the coordinate s (for simplicity of description, only one dimension is considered; a two-dimensional model is shown in section 6.4.1), and we assume the following components for the dynamic behavior.

Translation. A time varying translation of an x_i discrete pattern can be modeled by the term T in expression (6.5). Function $f_T(t)$ has only the values 1 or 0; the ratio of occurrence of these values determines the velocity of the movement; the sign at the increment of i determines the direction of the movement.

Linear growth or dissipation. Such a behavioral component can be represented by the term L in expression (6.5). Function $f_L(t)$ is simply a coefficient whose value may vary in time. The operator ξ is a test that disables this term whenever x_i is below a given threshold (the noise value), and above a certain value (the maximum echo intensity that is realistic).

Exponential growth or dissipation. A discrete approximation of a term of the form e^{kx} is represented by the term E in expression (6.5). f_E is a coefficient whose value (very small) may vary in time. The operator ξ has the same function as previously indicated.

Diffusion. From the conventional expression of the heat diffusion

$\frac{\partial u}{\partial t} - k \frac{\partial^2 u}{\partial x^2} = 0$, the finite difference approximation represented by term D in expression (6.5) is derived. This term provides a very versatile control. Function $f_D(t)$ is a coefficient that varies in time. When it is negative, different degrees of diffusion can be obtained. When it is positive, and sufficiently small to keep the system in stability, at least for a certain time, a sort of concentration is produced, and at every variation of the first derivative of x , a new peak grows. Coefficients $h(t)$ and $k(t)$ assume only integer values, and normally are equal to one; when they are not equal, asymmetric diffusion is produced; if they have large values, new distant cells are produced; and if they keep large values for a certain time, waves can be generated from a single initial cell.

$$\begin{aligned}
 x_i^{n+1} = & \overbrace{f_T(t)[x_{i+1}^n - x_i^n]}^T + \overbrace{\xi f_L(t)}^L + \overbrace{\xi f_E(t)x_i^n}^E + \\
 & \underbrace{+ f_D(t)[x_i^n - \frac{1}{2}x_{i-h(t)}^n - \frac{1}{2}x_{i+k(t)}^n]}_D
 \end{aligned} \tag{6.5}$$

Expression (6.5) can be computed at each point of the radar range by an abstract machine with the state diagram of Fig. 44. In this program, at first, an echo intensity is determined at each point from the actual radar echoes. These measurements are then used as initial values of the x_i in expression (6.5). Subsequently, the iterations proceed, with the f_i values manipulated by the operator on switches -- this is only phase (a) as previously defined. The profile generated by the present x_i values is made visible in an analog display, and the iteration frequency can be varied in order to slow down the evolution, and also freeze a particular stage in time. Different types of evolutions can be produced and then compared with the actual echo patterns detected by the radar at later times.

After the previous examples, the state diagram of Fig. 44 should need only a minimum of verbal description. FSM 1, the control, commands the initial data acquisition by means of states 1 and 2. In state 3, a command of the operator gives the go ahead for the running of the numerical model. State 4 starts the execution of the terms. State 5 commands the execution of term T; this state is transitted $t/100$ of the iterations, thus the parameter m allows the operator to control the speed of the movement in hundredths. State 6 is a waiting state for controlling the frequency of the iterations through the parameter t . FSM 2 and FSM 4 produce the boundary values in the segment of x_i . In FSM 3, state 5 shifts the x_{i-h} , and state 6 shifts the x_{i+k} . State 7 implements the D term, while the two tests implement the operator ξ . State 8 implements terms E and L.

This program has been adapted for the CPL 1 machine, and it is contained on two punch cards. An exercise with this program was previously shown in Fig. 23 as breaking waves obtained by an initial single cell.

Comparison with equivalent programs written in different programming languages is given in Table 11.

Section 6.2.3 described how to compound a program from several different ones. Chapter 5 discussed how complex programs are developed as an interplay of FSMs and pages. One can think of assembling a compounded program that (a) determines continuously in real-time certain characterizations of the present echo patterns continuously detected by the radar, (b) reads certain coefficients set by the operator on the basis of general information, and (c) produces continuously on suitable displays a set of future echo patterns in accordance with different hypotheses.

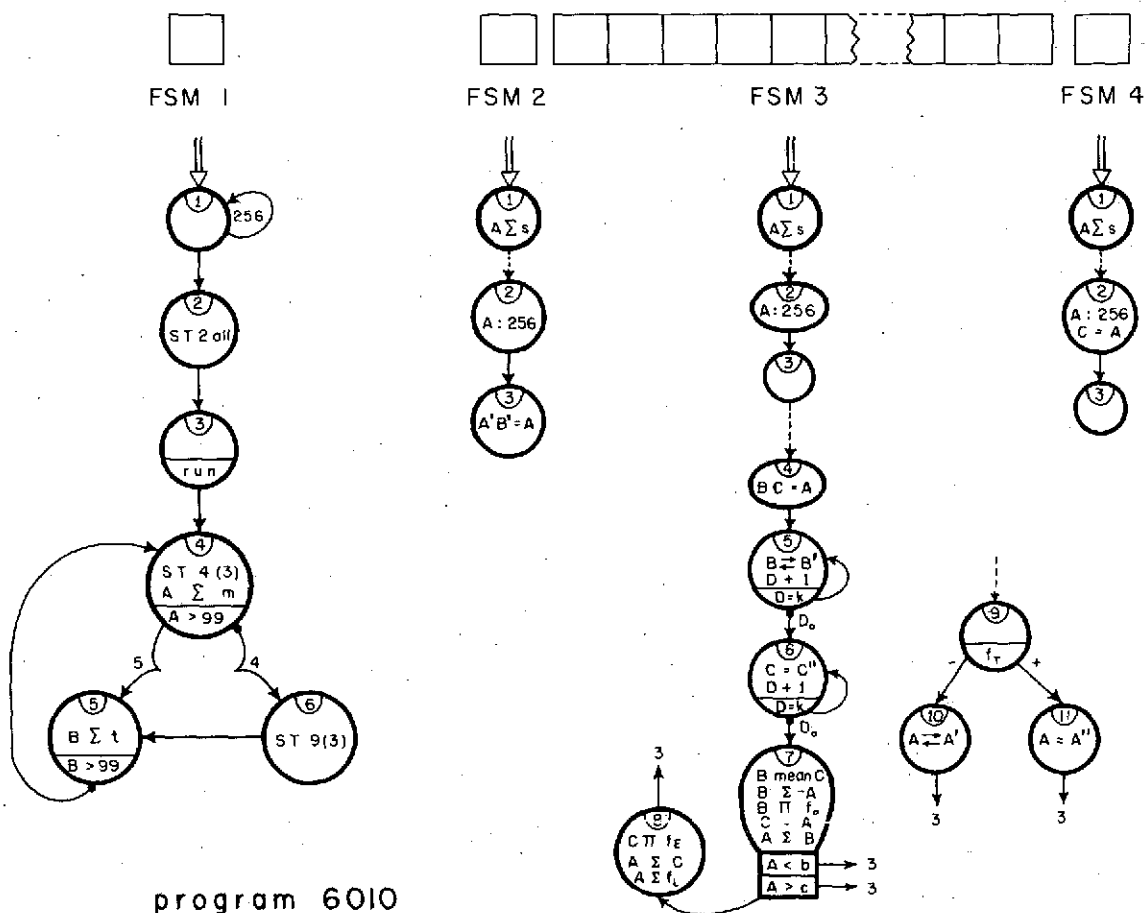


Fig. 44 - State diagram of a real-time numerical model

T A B L E 11						
program 6010						
language	number of statem./instruc.	number of symbols	number of cards	memory words	execution time per segm.	programming time
C P L 2	54	208				
C P L 1	71		2	152		

for definitions and criteria see section 6.1.3

6.2.11 Computation of the Fast Fourier Transform

Doppler weather radars have the potential for measuring wind velocity and turbulence in the atmosphere (see Atlas 1964). In these measurements, an independent Fourier transform is required at each point explored by the radar. To avoid a prohibitive recording and playback systems, with the consequent delay in gathering the information, all these Fourier transforms should be executed in real time. Indeed, such an application is a good bench test for the processing power of a computer. In the following, a program that performs such a task is described for a CPL system*, using a fast Fourier transform.

A sequence of time samples f_n ($n = 1, 2, \dots, N$) admits a discrete Fourier transform composed of N frequency samples F_k given by

$$F_k = \sum_{n=1}^N f_n W^{nk} \quad W = e^{-j\frac{2\pi}{N}} \quad (6.6)$$

Such a transform requires the computation of N^2 complex terms. The fast Fourier transform (FFT) is a class of algorithms that obtains the transform (6.6) with drastically reduced need of computation by taking advantage of certain regularities in the transform itself (Gold et al 1969). In the FFT, computation is essentially reduced to the basic complex operation (called butterfly from its structure), with P , Q , and W complex quantities,

$$\begin{aligned} P(i+1) &= P(i) + Q(i)W^Z \\ Q(i+1) &= P(i) - Q(i)W^Z \end{aligned} \quad (6.7)$$

performed repeatedly on a specific pattern of data. Such patterns are best shown in the form of data flow graphs, one example of which is in Fig. 45. for a transform of 16 points. A property of the pattern of Fig. 45 is that each pair of new terms can substitute orderly ("in place") the old terms from which they are computed. The pattern has also a typical shuffling of the resulting frequency samples. Patterns are possible in which also the final frequency samples are in the natural order. The number of terms to be computed in the FFT (involving complex multiplication and complex addition) is $N \cdot \log(N)$. Because these computations are all in pairs that use common

*This program has been derived from the report "Study of the Applicability of the CPL System to Doppler Radar Signal Processing" (CS 11-73) prepared for the National Center for Atmospheric Research, Boulder, Colorado.

terms, and each pair can be computed simultaneously (butterfly), the number of actual operation cycles is $1/2 \cdot N \cdot \log(N)$.

By comparing the structure of expression (6.7) with that of expression (3.2) at page 79, and by interpreting the data flow of Fig. 45 as a periodical rearrangement of page flow, it can easily be deduced that the FFT is a natural computation for a CPL system. Moreover, because the number of pages does not affect the complexity of the FSMs, independent FFTs can be simultaneously performed at adjacent points in range.

The class of FFT for radar time sequences, repeated independently at different range points, can be implemented by a CPL machine as follows. The memory is partitioned in channels, one for each time sample; in each of these channels, the range sequence of samples (belonging to a given time sample) are stored. Each computing page is composed with the data from two channels; the choice of these channels in the course of the computation depends on the chosen FFT algorithm. The data of the page after each computation cycle substitute for the old data in the same channels. In this way,

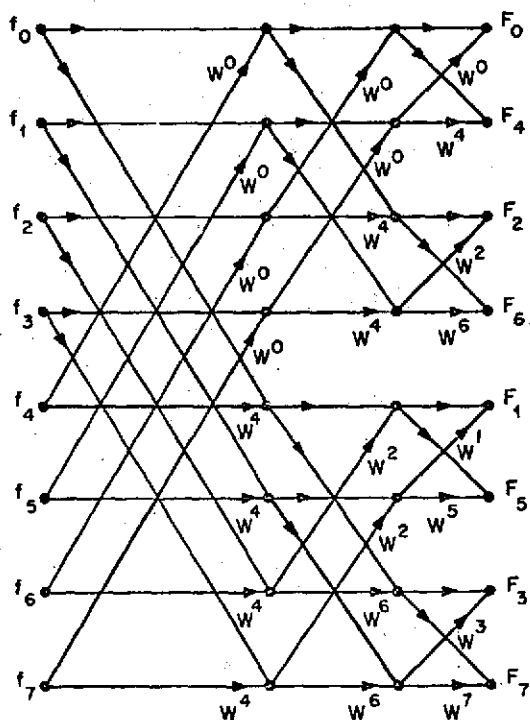


Fig. 45 - Data flow in a FFT algorithm

the minimal storage capacity is used, which is N complex words for executing an FFT of N points. For a given memory capacity of the CPL processor, different choices of points in range and of points in time can be chosen through programming. If M is the total number of complex words in the memory, N the points in time, and R the points in range, assuming that all are binary numbers, the choices are given by $N \times R = M$.

These pages implement an FSM that describes the computation of expression (6.7). Another FSM, implemented by a separate page, computes the coefficients W^Z in expression (6.7). An additional FSM, implemented by still another page, controls the entire process, in particular the selection of the channels for the data pages. The details of these FSMs depend on the particular form of the FFT adopted. As an example, here, an FFT with decimation in time, scrambled output, data in place (corresponding to Fig. 45) is considered. The state diagram for the FSMs is shown in Fig. 46.

FSM 1 (control)

In state 1, n circulations elapse for the acquisition of the time samples. In state 2, a driven transition to state 2 for all pages is produced, and A, B, C, D are used for producing the particular pattern of numbers that constitute the labels of the data channels. There is a periodicity of decreasing duration in state 3 (corresponding to the periodicities visible in Fig. 45), and in states 2 and 4 these numbers are replaced. At each cycle, this page routes two channel labels to the control of the memory. At each passage through state 4, a driven transition to state 3 is directed to FSM 2. At the end of $\log(N)$ series of computations, the page transfers to state 5; here a driven transition to state 3 is directed to FSM 3 for the output production, and, in state 6, bit-reversed numbers (function available in hardware) are routed as channel labels. After the output of the N series of frequency samples, the process ends. Of course, different arrangements for the output can be prescribed; for instance, the output can be simultaneous with a new input acquisition.

FSM 2 (coefficients)

The complex coefficients W^Z are stored in $C'D'$, where they are read by the computing pages. The starting value $(0, 2\pi/N)$ is produced in state 2; the series of the following values are produced in state 3 (the sine and cosine functions are available as look-up tables).

FSM 3 (butterfly computation)

This FSM is implemented by one page per point in range. In state 1, the input complex samples (re, im) are acquired to form the content A B of the successive pages. In state 2, the complex numbers (A,B) and (C,D), from the channels prescribed by FSM 1, are operated upon as in expression (6.7). The coefficients W^Z are in C'D', and the butterfly operation is available as a special function of type 4 (section 5.2.3). In state 3, the complex frequency samples (A,B) are normalized and routed to the output.

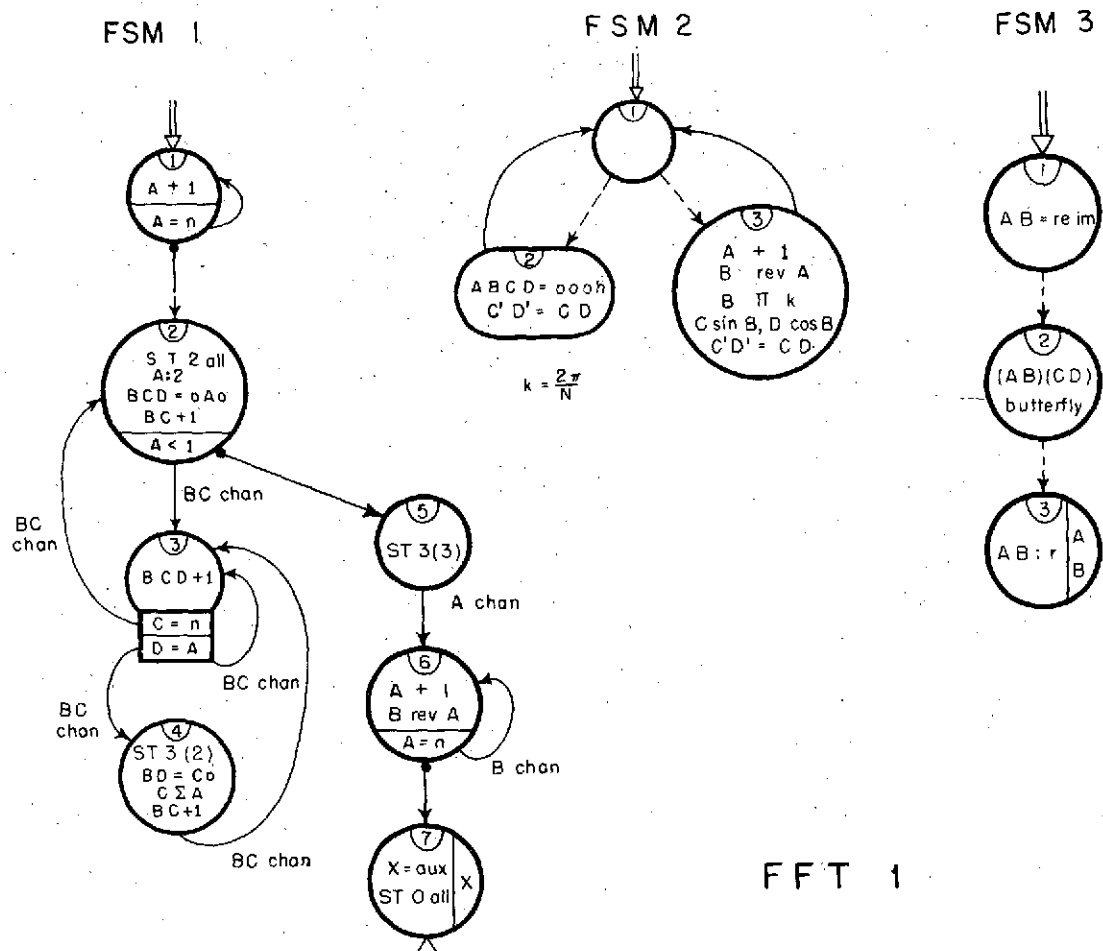


Fig. 46 - An abstract machine for performing the fast Fourier transform simultaneously in a sequence of points

6.2.11

The data of this program that are relevant to the comparison of size and complexity are reported in Table 12.

T A B L E 12 program F F T 1						
language	number of statem./ instruct.	number of symbols	number of cards	memory words	execution time per segm.	programing time
C P L 2	44	209			r t	
for definitions and criteria see section 6.1.3						

6.3 REAL-TIME PROCESSING OF METEOR-RADAR SIGNALS

The first use of a CPL processor was in connection with a radar station for the study of the ionized trails produced by meteors in the high atmosphere, at the Smithsonian Astrophysical Observatory (see section 4.4.1). In this context, a variety of programs for pattern recognition, statistics, and measurements were developed. A few samples of them are reported here as a further illustration of the programming language.

6.3.1 - Recognition and recording of faint meteors

The purpose of this program is to recognize faint meteors (echo typically below the noise) and produce periodically on magnetic tape tri-dimensional matrices of the detected meteors (echo energy, echo duration, and range). Fig. 47 shows a sample of printout from a recorded tape showing time and data as heading, a row of auxiliary data, and the number of detected meteors versus energy (horizontally) and duration (vertically) for a particular range segment and time interval.

The meteors of interest produce a narrow echo that rises in intensity in the first few pulses and then decays exponentially. Ionospheric echoes of a different pattern, and all sorts of interferences should be rejected as much as possible. Fig. 48 shows one among the several abstract machines that were devised at that time for accomplishing this task. The circulation of the CPL system is synchronized with the pulse emission of the radar, and a sequence of pages (one per width of the expected echo) performs the FSM described by states 0 through 6. Twenty consecutive samples (sam) of the radar video are integrated and called energy (E_n). In order to start this integration approximately at the beginning of the echo, a tentative accumulation (A_c) of five samples is continuously made and compared with a threshold (t_1 , in state 1). If the accumulation is less than the threshold, the cycle starts new in state 0; if larger, a count (C_n) is increased by one (state 2). If four consecutive accumulations are above the threshold, the process transfers to state 3 which, in conjunction with state 4, determines the duration in intervals of 20 pulses. For each sequence of 20 samples that exceeds a threshold t_2 , the energy is increased by a number h such as to produce the shift of one row in the output matrix. Every time the

accumulation of 20 samples does not exceed the threshold, or the last row of the matrix is reached ($E >$), transfer is made to state 5 which normalizes the energy and routes it to the distribution. If at any time an echo is recognized also at the adjacent range (notified through the signal S in state 3), the echo is disregarded by means of a priority transition to state 6, where the echo is tracked until it completely disappears for 128 pulses.

Simultaneously another page performs the FSM of states 7, 8 and 9 for producing statistical measurements of the noise by accumulating in C one thousand samples, and in D the times in which the noise exceeds a value e . These measurements are made available to the record FSM through Ω'_N . The record FSM (state 10), as soon as initiated, acquires the several parameters and measurements involved in the process, produces a record, and disappears (triangle attached to the state).

Another page executes an algorithm (state 12) for testing continuously the computation of interest and produces a diagnostic output (state 13) every time a computation is incorrect. Another page reads information set on switches by the operator (state 11) and at the command β gives it to the output. State 13 is used by both the last FSMs through stopover transitions.

The program described in Fig. 48 in the user language, when translated into the machine codes, for the CPL 1 processor, is contained on two punch cards. Another card is necessary for the supervisor program which schedules the process in time and range.

This process has been written also in PL1, Fortran, and assembler language for the 370/155 of the Information Processing Center at M.I.T. The flow chart of Fig. 49 shows the algorithms applied to the vectors in which the data of the problem are organized. Table 13 summarizes data on the different programs. The program of Fig. 48, in its original version indicated as CPL 1, was written debugged and put in operation in a single night during a week of continuous recording of faint meteors at the Smithsonian radar station. The PL1 version was accomplished in several days.

The FSMs described in Fig. 48 provide the complete processing, including data acquisition and recording on the output tape. In the flowchart of Fig. 49, the operations of data acquisition are excluded; the routines involved in the production of output records are not indicated. In Table 13, separate account is given for the algorithms alone, and the entire process inclu-

Fig. 47 - A record on the magnetic tape

0	10 SEC	30 MIN	12 H	11 O	03 MON	1971	287	4214 PROGRAM	4000 SUP		
0		23	45	10	0		0	0027	0144	0000	0437
0		45	2	7	2		2	0055	0012	0002	0000
0		2	0	0	3		1	0002	0007	0004	0002
0		1	2	2	7		4	0002	0000	0005	0001
								0001	0002	0007	0004

program 4214

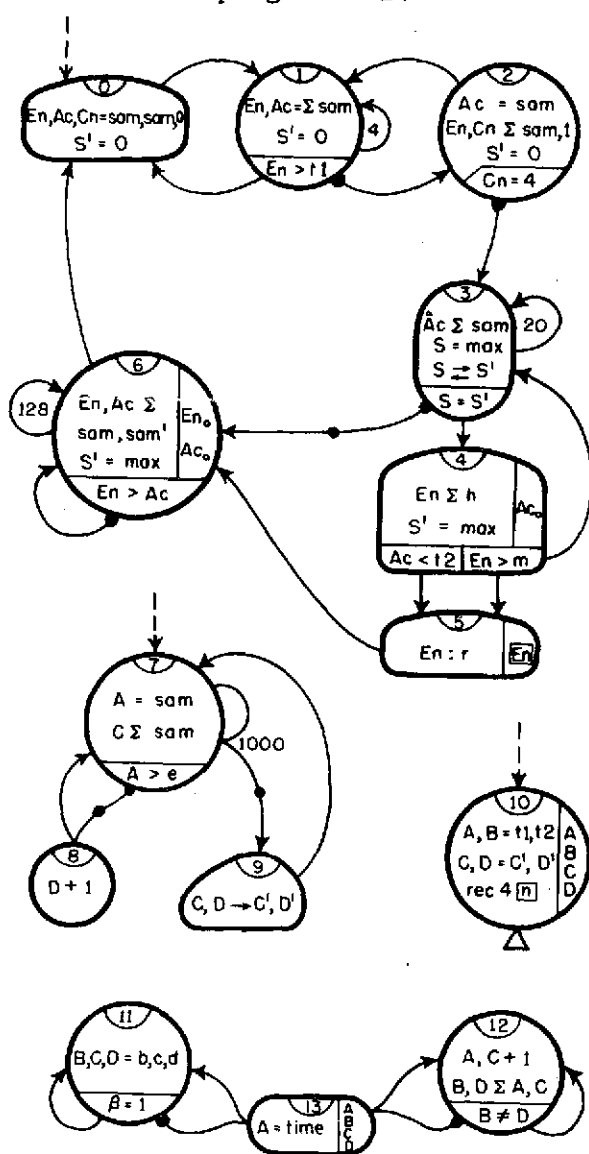
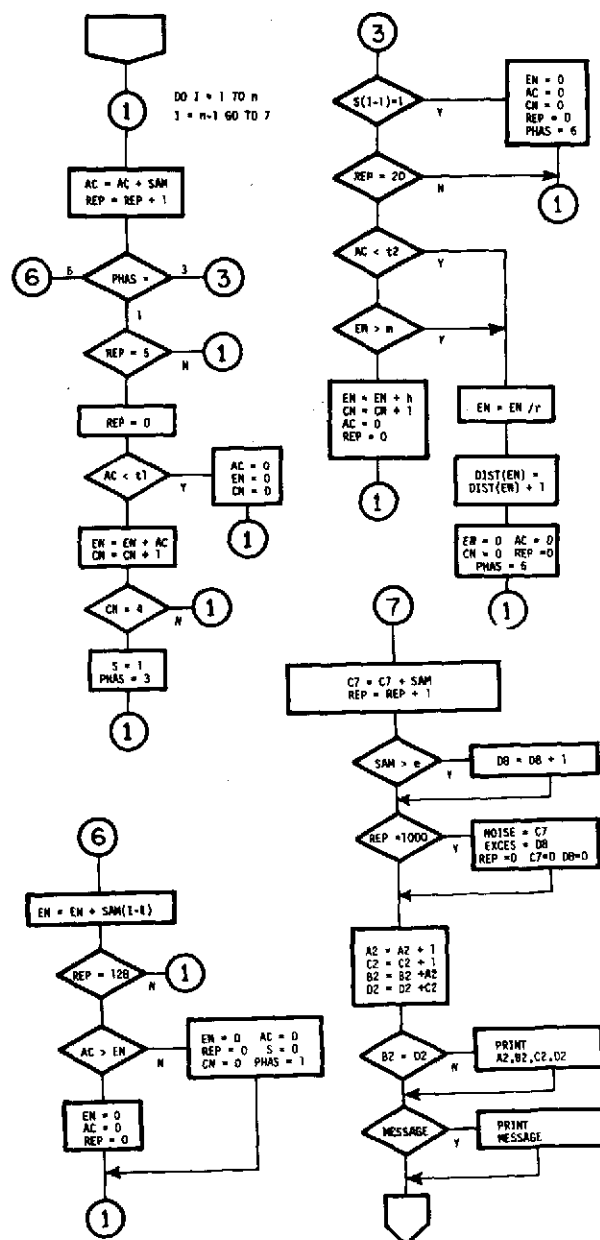


Fig. 48 - An abstract machine for the detection and recording of faint meteors



6.3.1

sive of input and output operations. For each program, the input data are supposed to be recorded on magnetic tape, in the format recommended for each of the languages used.

In the evaluation of the number of symbols in the source programs, one symbol has been accounted for each name, constant, arithmetic symbol, and punctuation mark (including significant blanks). For the program of Fig. 48, its equivalent version in the form of strings of symbols has been used; but, in this case, each item is accounted for a number of symbols equal to the number of bytes necessary for its expression.

The optimizing compiler optimizes execution time at the expense of compilation time and memory size. The execution time is average in the case of the conventional computer.

TABLE 13 - Data related to the program for faint meteors.

Language	source program				object program					execution time per radar pulse (μ s)
	algorithms alone		complete process		algorithms alone		complete process			
	number of statem.	number of symbols	number of statem.	number of cards	number of instr.	memory bytes	number of instr.	number of cards	memory bytes	
P L / 1	93	662	150	199 §	338 *	1352 *	1162 *	139 *	4588 *	264 *
					171 Δ	684 Δ	690 Δ	106 Δ	2682 Δ	68 Δ
									6928 * ✓	
									5038 Δ ✓	
FORTRAN IV	73	698	116	116	223	886	427	55	1678	46
ASSEMBLER	138	1029	434	437	138	560	704	45	1738	22
									4116 ✓	
C P L 1	69	265	137	3	69	320	137	3	480	6.66
C P L 2	48	186	116	2	48	160	116	2	1046 ✓	
									320	886 ✓

* compiler PL/1 (F). Δ PL/1 Optimizing Compiler, IBM, 1972.

✓ including also data storage for a segment of 50 range elements.

§ for complex and label statements, more than one card were used to ease reading and debugging.

For this process, a more through comparison has been made among the equivalent programs in different languages, and the occasion is taken for some discussion.

The fact that 690 instructions are needed by the computer, compared with the number of sentences that are necessary for describing the process shows the distance between point A and C in Fig. 1. The need for 150 statements in PL/1, compared with the 690 machine instructions, shows the well-known power of a high-level language (point B in Fig. 1). The 690 instructions produced by the optimizing compiler, in respect to the 704 produced manually at the assembler level, show the achievable effectiveness of an automatic compilation. The 1162 instructions of the compiler F testify to the difficulties involved in the compilation process.

The 137 statements of the CPL 1 or the 116 of the CPL 2 show that the approach of the abstract machines has a power not inferior to a high-level language, at least for the process here analyzed. The one to one correspondence between statements for the abstract machines and codes for the CPL system shows that a unique representation of the process for the user and the machine is feasible.

The fact that the CPL processor used is a rudimental and limited machine of this type is of little account in our discussion. In this language, the length of the words tends to increase with the logarithm of the number of options in the hardware. On the other hand, the complexity of the problem models decreases with these options. The syntactic structure of the FSMs does not change with the size or complexity of the system. Thus an increase of complexity of the computer does not imply an increase of complexity of the programs. In this respect, the comparison of the CPL 1 and CPL 2 programs is relevant; the two machines are the same size, but the CPL 2 exploits to a larger extent the features of the hardware and of the corresponding language.

This approach implies that we can generate as many functions as we like and express them with a specific word, as long as we are able to conceive a network that produces that function. In other words, the flexibility achieved in conventional programming by designing routines here is obtained also by designing operating structures. Of course, sequentiality can always be used, either because it is inherent in the function at hand, or because of limitations in the actual PN available.

The often claimed man-machine interaction needs a careful interpretation. The machine does the 690 commands (in our example); the interaction is with a ghost machine, that of the 150 statements; therefore we are limited to the capability of the available translation means. An interaction by means of a common modeling, like that of Fig. 48, is more honest and thus has more chance to be effective. The factors that make programs so concise can be viewed as follows: (1) modeling the problem in the form of an FSM leads to a very concise description of the problem; (2) describing the functions of the problem in the form of operating networks, and using highly inflected words for describing the networks, leads to a very concise description of complex functions; (3) the greatly reduced computer overhead eliminates most of the numerous computerrelated statements necessary in conventional programing; (4) the language used for describing an entire program allows much greater exploitation of the information capability of strings of symbols.

It should be clear that Fig. 48 is not a flow-chart in the usual sense of graphical documentation of a program already written in a phrase language, as Fig. 49 is. Fig. 48 is the abstract machine that solves our problem and constitutes the original complete program; the equivalent representations in the form of strings of symbols are derived afterward from it, as in the example shown on page 96.

It is interesting to note that the structure of Fig. 48, in spite of the fact that it has been introduced as a machine (abstract), turned out to have less elements not existing in the user view of the process (such as indexes and computer related steps) than the structure of Fig. 49, which is introduced as algorithm of the process. Undoubtedly, the flexibility of the FSMs makes them closer to our inner pattern of thinking than each single programing language. In using this system, we discuss the problems in terms of states and transitions, we punch the program cards in terms of those problem states and transitions, and, with more ease than in conventional programing, we debug in the same terms. We feel that problems are not deformed by the machine, but the machine forms itself after the problems (Schaffner 1972a).

6.3.2 - Experiments of strategies

The recognition of faint meteors is affected not only by their intensity relative to the noise intensity, but also by the strategy used for their recognition. If the meteor echoes were constant signals in a white, Gaussian noise, a matched-filter receiver would be the optimum system for their detection. In reality, there are man-made interferences that are more prominent than the cosmic noise, and the meteor echoes appear as short bursts with an unpredictable time behavior. Therefore, recognition strategies have to be devised, in terms of the possible characteristics of the meteor echoes, and in terms of the parameters that are to be measured simultaneously to the detection of the meteors.

The modeling of the processes in the form of abstract machines facilitates significantly the development of such strategies. Moreover, the fact that the computer performs the processes in the same way as the user conceives them allows an easy verification of the effectiveness of the strategies by observing the values of the variables involved during the execution of the processes on the actual signals.

6.3.3 - Measurement of noise

The detection of faint meteors is on a statistical basis. Essential for these statistics is a precise knowledge of the characteristics of the noise present at each moment. Several programs were prepared for recording statistical parameters on noise and interference, automatically at periodical intervals of time.

One program is described here as a simple example of the flexibility and conciseness of the language. One thousand noise samples s are analyzed. The distribution of their amplitude should be produced. Moreover, the number of samples that exceed a threshold t , and their mean amplitude should be computed.

Two versions of this program are given: one (Fig. 51a) for a small page and a large recourse to functional memory; and one (Fig. 51b) for a larger page and a minimum recourse to functional memory. It can be noted in Table 14 that the larger page leads to a reduced description of the program. These considerations are relevant to the question of the complexity of the program-able network in a CPL system.

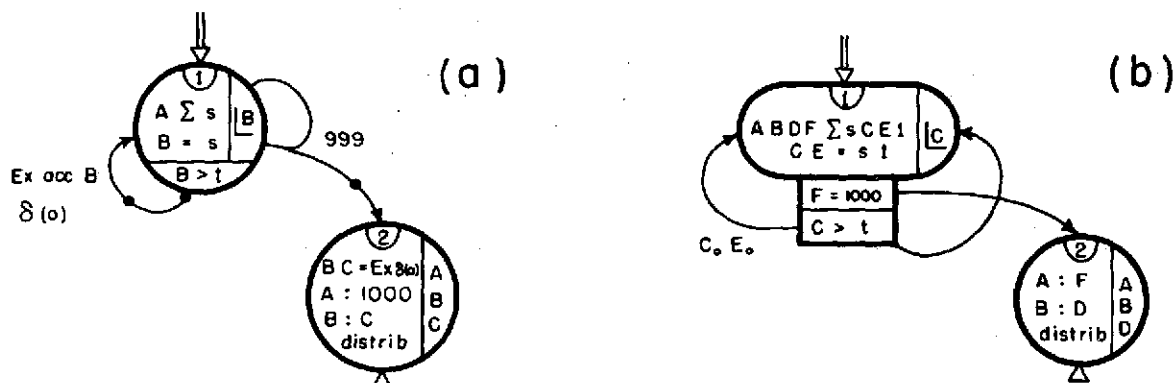


Fig. 51 - State diagrams for measurement of noise characteristics

T A B L E 14				program 1020		
language	number of statem./instruc.	number of symbols	number of cards	memory words	execution time per segm.	programing time
C P L 2 (a)	13	61	1		r t	
(b)	11	59	1		r t	

6.4 EXPLORATORY PROGRAMS

To explore further the effectiveness of the programming language described in section 5.2, and the efficiency of the corresponding execution, some programs for more complex processes are examined in this section.

6.4.1 - Numerical model of the dynamics of a fluid

In section 6.2.10 preliminary experiments on numerical models were shown. Numerical models constitute a class of processes in which the spatiotemporal structure is basic; therefore, we can expect that their modeling in the substratum of chapter 3 should in general be particularly efficient. In this context, a program for a two-dimensional model, written in the language of chapter 5, is shown in this section.

The dynamics of a hypothetical fluid is modeled in the form of an initial-value problem with boundary conditions. The analytical expressions considered are

$$\begin{aligned}
 \frac{\partial \eta}{\partial t} &= h_1 \frac{\partial \eta}{\partial x} + h_2 \frac{\partial \eta}{\partial y} \\
 \frac{\partial \psi}{\partial t} &= h_3 \frac{\partial \psi}{\partial x} + h_4 \frac{\partial \psi}{\partial y} \\
 \frac{\partial v}{\partial t} &= h_5 \frac{\partial v}{\partial x} + h_6 \frac{\partial v}{\partial y}
 \end{aligned} \tag{6.8}$$

where η , ψ , and v are the variables of the system, and h_r the given parameters. The chosen finite-difference approximation is given by

$$\begin{aligned}
 \eta_{i,j}^{n+1} &= \eta_{i,j}^n - k_1 (\eta_{i,j}^n - \eta_{i-1,j}^n) - k_2 (\eta_{i,j}^n - \eta_{i,j-1}^n) \\
 \psi_{i,j}^{n+1} &= \psi_{i,j}^n - k_3 (\psi_{i,j}^n - \psi_{i-1,j}^n) - k_4 (\psi_{i,j}^n - \psi_{i,j-1}^n) \\
 v_{i,j}^{n+1} &= v_{i,j}^n - k_5 (v_{i,j}^n - v_{i-1,j}^n) - k_6 (v_{i,j}^n - v_{i,j-1}^n)
 \end{aligned} \tag{6.9}$$

with the conventions:

$$\begin{array}{ll} x = i \Delta x & i = 1, 2, \dots I \\ y = j \Delta y & j = 1, 2, \dots J \\ t = n \Delta t & n = 1, 2, \dots N \end{array}$$

and the k_r derived from the h_r .

To obtain the solution, an abstract machine is conceived, that has a page for each point of the two dimensional space of the system (Figure 52), a page for each boundary point, and a control page. The symbols η , ψ , and v , related to the variables of the process, are considered as names for three variables x_r ; initial values a , b , and c of these variables are treated as input data u_r ; the parameters k_r also are treated as input data. Moreover, an additional three variables x_r , named D , E and F , are used for temporary purposes. The pages related to the points of the fluid perform an FSM 3 as described in Figure 53, which implements expressions (6.9); the pages related to the boundary points perform an FSM 2 which implements a time evolution of the boundary values; and the control page performs an FSM 1 which controls the work of the entire system. The pages circulate in the structure of Figure 13, with different scanning as indicated in Figure 52.

Figure 53 should convey the level of abstraction of these operational structures. For instance, FSM 1, which constructs and controls the entire machine, has four states. State 1 is devoted to creating the page array indicated in Figure 52. In this state, function F consists simply in incrementing variables A and B by one. Function T is expressed as a self-explanatory decision table (the transition from the corner corresponds to the "else" condition). The routing is different for the different transitions and consists of creating pages related to given FSMs and in clearing variable A . State 2 prescribes a horizontal scanning of the pages. State 3 prescribes a vertical scanning, and provides for the test of the number of time steps. State 4 orders an output record of the computed quantities, and makes the pages disappear (transition to a triangle).

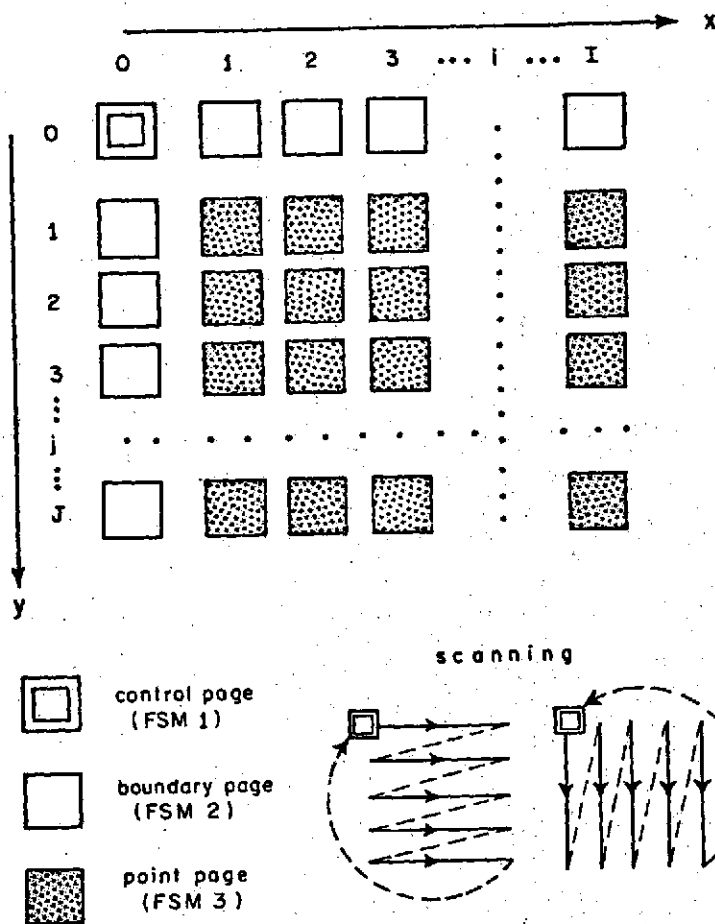
The computation of the variables η , ψ , and v at each point (FSM 3) is obtained by means of simple networks of a parallel nature established by the

6.4.1

user in accordance with expressions (6,9) As an example, in state 1 of FSM 3, the input set U , consisting of the data a , b , and c , is transferred in parallel into the set X , consisting of the variables η , ψ , and v . In state 3 of FSM 3, there is a succession of five networks: the first produces simultaneously the accumulation of the original values of D , E , F into η , ψ , v , and the transfer of the original values of η , ψ , v into D , E , F ; the second produces an interchange of values between E , E , F and D' , E' , F' , which are variables that remain in Ω'_N of the network during the circulation of the pages; the third produces the subtraction of D' , E' , F' , from D , E , F ; the fourth produces the multiplication of D , E , F by the data set k_2 , k_4 , k_6 ; and the fifth the accumulation of the present values of D , E , F into η , ψ , v . A routing prescription sends the present values of η , ψ , v to an output storage.

Obviously, the interest for such constructs is not to make the user do what can be provided by a compiler, but to give the user the possibility either of providing what has not been anticipated by the software systems, or of obtaining specific optimizations. In this example, the aim was to minimize the memory and the execution time. The entire computation is made with $6IJ + 3(I+J) + 2$ memory words. The machine cycles are $(2N+2)(I+1)(J+1)$, with an average of four to five networks per cycle.

Fig. 52 - Two-dimensional page structure



REPRODUCTION OF THE ORIGINAL PAGE IS POOR

FIG 3

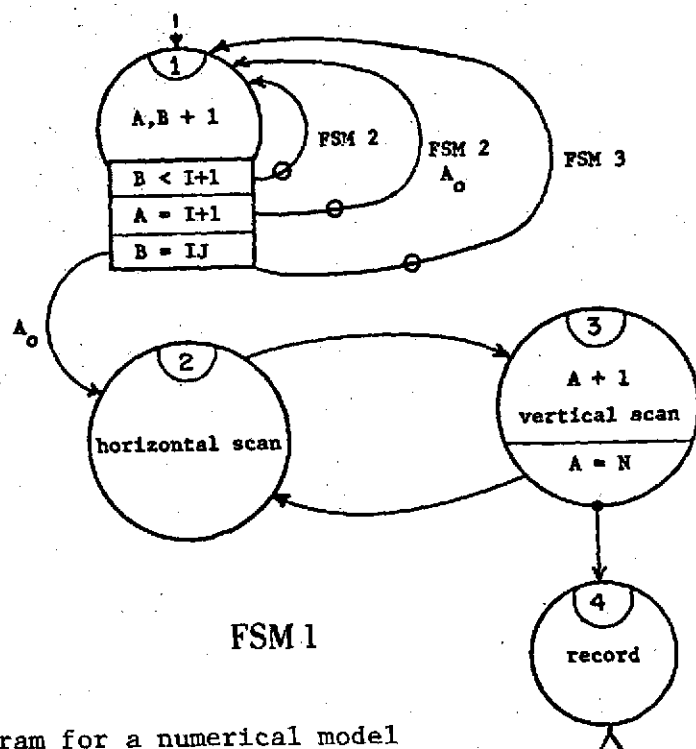
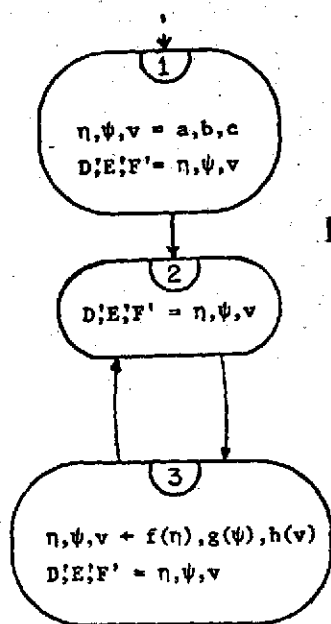
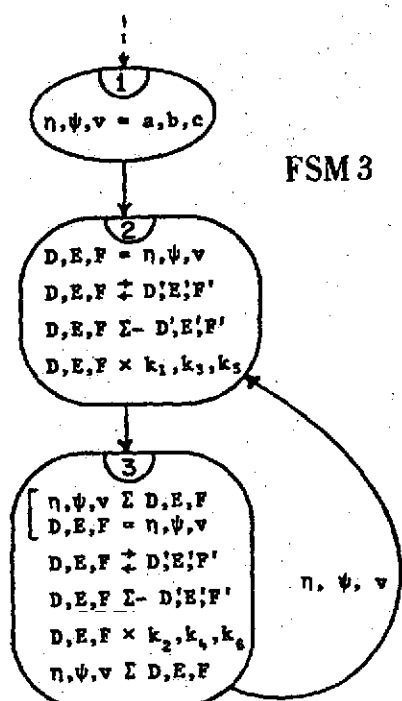


Fig. 53 - State diagram for a numerical model

6.4.2 - Analysis of echo-pattern turbulence and movement

In this example, a hypothetical process for weather radar is used for exercising the flexibility of the abstract machines as models of intriguing data manipulations.

The objective of the process here described is to infer on wind distribution in the atmosphere from the movement of radar echo patterns. Essentially, profiles of the echo intensity along given lines s are measured at two times differing by Δt (Fig. 54); then the two profiles at each line s are cross-correlated and the Δs for which the correlation is maximum is determined. The ratio $\Delta s / \Delta t$ is assumed as an estimate of the mean apparent movement of the scatterers along s . The echo intensity is determined in an extended volume for adjacent small cells forming a cartesian three-dimensional array, and cross-correlations are performed along three orthogonal axes for all adjacent lines of cells. Because the radar scans space in polar coordinates, a coordinate conversion is necessary in order to attribute the arriving echoes to the proper cartesian cells. We are interested in both the local distribution of movement and the global value for the entire volume.

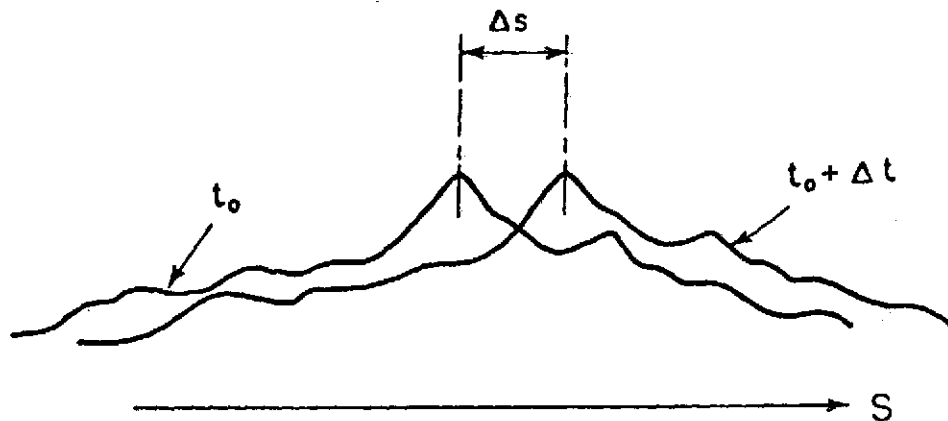


Fig. 54 - Echo profiles to be crosscorrelated

An abstract machine for implementing such a process is derived from the visualization of it on the part of the user. First we establish the data structure. Three orthogonal coordinates r , t , and h are considered (Fig. 18 in section 5.2.1). Pages CE (cell pages), organized in a three-dimensional array, are assigned to the data and computations related to the cells of the volume considered in the atmosphere. Pages TE (terminal pages) are allocated at the ends of each row and column of pages CE; thus there are three planes of pages TE, one for each coordinate. Three pages AV (average page) are assigned to the collection of the data from the three planes of TE pages. A linear array of pages IN (integrating pages) are related to the points of the present radar sweep. Finally, one page CO (control page) is assigned the coordination of the work of all pages. Each type of the above pages implements a particular FSM, which will be designated with the same abbreviation (Fig. 55).

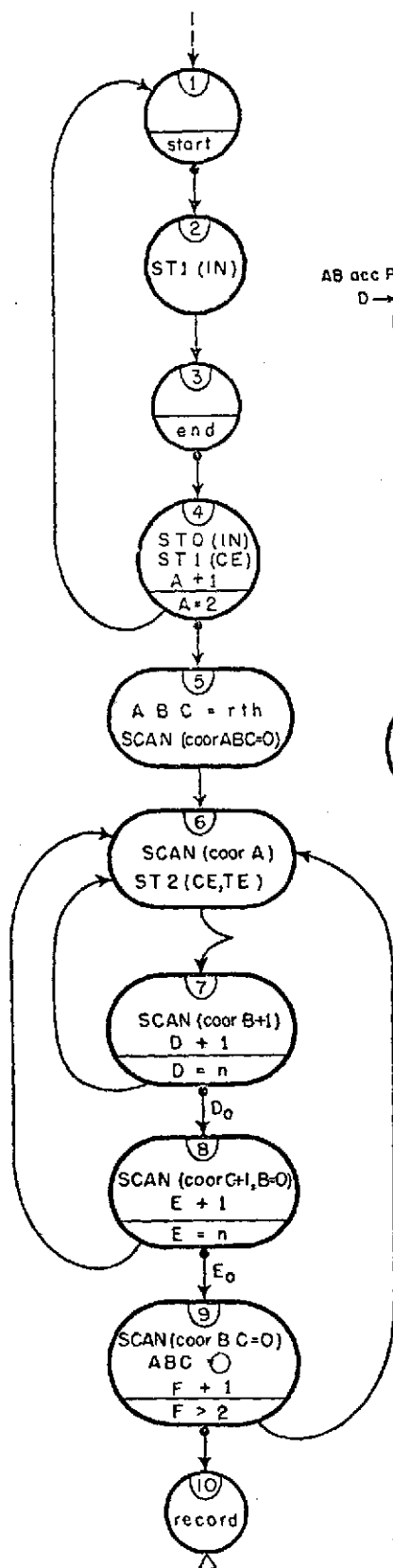
FSM IN

The task of this FSM is to integrate the echoes emerging from the radar receiver and to send them into the corresponding CE pages, thus providing the conversion from polar to cartesian coordinates. More precisely, we want all the independent echoes that fall into each cubic cell be integrated in space and time, and then delivered to the corresponding CE pages.

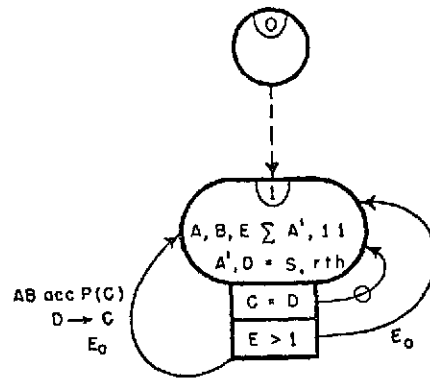
Such a computation can be tackled in several different ways. Here it is shown that an abstract machine can perform this task in real time with one state. The antenna beam crosses several adjacent cells at each time; and during the movement, the beam abandons certain cells, and intersects new cells. In accordance, we imply a sequence of pages in number not less than the number of cells that can be crossed at any time. These pages have operating cycles at the rate at which the echoes become independent (say, corresponding to the radar pulse width). In each cycle, in the state of FSM IN (Fig. 55), variable A accumulates the present content of A'. At the same time, B and E are incremented by one, and D acquires the present coordinates r , t , and h condensed in one word and varying by one unit at each cell. Variable C holds from some previous cycle the coordinates of the present cell.

At each cycle, a double test is made. Until $C = D$ (i.e., present coordinates are the same as they were previously), the page remains in PN (transition with circle), and thus A accumulates samples consecutive in range. As

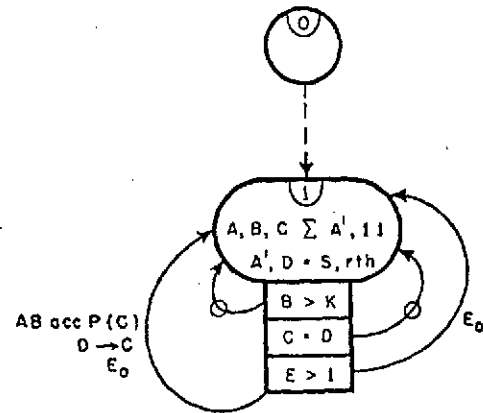
FSM CO



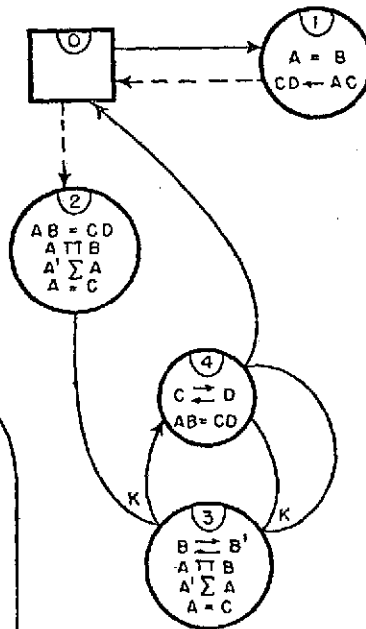
FSM IN



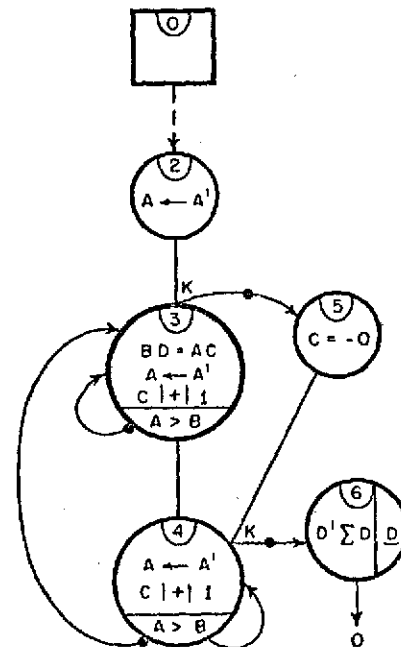
FSM IN'



FSM CE



FSM TE



FSM AC

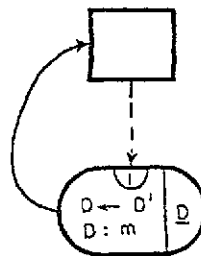


Fig. 55

State diagrams of
concurrent FSMs

soon as any one of the coordinates changes, the page enters circulation, and the present sample in A' will be taken by the following page. Normally E is larger than 1, and thus the transition path that clears E is taken. At the next circulation, the page will perform the same way. But the time will come when that cell is no longer crossed by the antenna beam, and thus at the next circulation the coordinates will be different at the first cycle; in this case, E is no longer larger than one, and the transition path with routing is taken. This routing accumulates the content of A (the accumulated echo) and that of B (the number of accumulations) into the page with coordinates equal to C. Moreover, the present coordinates (presently in variable D) are transferred to variable C in the page.

Depending on the speed of the antenna movement, and on the range, the beam may cross the same cell for such a large number of times, that the accumulation of samples may exceed the capacity of A. The variant FSM IN' contains a further test by which the data are routed also when the number of accumulations in B exceeds a given value.

FSM CE

When in state 0, the pages rest in the functional memory, capable only of receiving the data routed by the pages IN. At the end of the first and second radar scans of the atmosphere, the pages CE are driven to state 1 by the control page CO. In state 1 the accumulation in A is divided by the number of accumulations in B, and the result is transferred into C while the previous content in C is transferred into D.

For the execution of the crosscorrelation, each row and each column of pages CE is circulated as a separate page segment. In state 2, A and B acquires the content of C and D, respectively (they correspond to points of the first and second profiles), and the product with zero lag is accumulated in A'. Then, in state 3, the profile in B is shifted k times, and the corresponding products are accumulated in A'. Subsequently, in state 4, the profiles in C and D are interchanged, and then the operations in state 3 are repeated again k times. Finally, the two profiles resume their original locations in C and D by stopping over state 4, and then the page segment returns to state 0.

FSM TE

When pages CE of a segment (either a column or a row of the array) transfer to state 2, the page TE of that segment also transfers to state 2 of its FSM (Fig. 55). In state 2, the accumulation of products in A' (a point of the crosscorrelation) is transferred into A of the page. Then the page stops k times over state 3 before transferring to state 6. In state 3, B and D acquires the new sum of products in A'; and C increments its absolute value by one. Every time the new sum of products in A' is not larger than the one presently in B, the page transfers to state 4, where the substitution of new values in B and D does not occur. In this way, D acquires a measure of the lag corresponding to the maximum sum of products.

In state 5, variable C is cleared and provided with the negative sign. Then the operations in states 4 and 3 are repeated. Finally, the page transfers to state 6, where the present value of D is accumulated in D' and also routed to the output.

FSM AC

At the end of the crosscorrelations along each axis, the corresponding page AC is circulated in state 1 in order to acquire the accumulations of lags in D'. A mean value is obtained by division, and then it is routed to the output.

FSM CO

The tasks of this FSM, implemented by one page, is to coordinate the work of the other FSMs and to produce a record of output data. Through states 1 to 4, under the control of signals "start" and "end" indicating the beginning and the completion of the radar scanning, pages IN and CE are driven to the proper states in order to produce the two sets of profiles.

After two radar scans, page CO transfer to state 5, where variables A, B, and C are given three indexes (say values 1, 2, and 3) related to the coordinates r, t, and h of the array of pages CE; moreover the scanning of these coordinates is made starting from zero. Then, in state 6, a page segment along the coordinate A is put in circulation, and the related pages CE and TE are driven to their state 2. After 2k circulations, coordinate B is incremented by one (state 7), and the scanning along coordinate A is repeated (state 6). After n increments of B (the size of the array), coordinate C is incremented

by one (state 8), and the scanning is repeated as above. After n increments also of coordinate C , the indexes of the coordinates are rotated one position (state 9), and all the previous phases are repeated. After two rotations of the indexes, page CO transfers to state 10 for the control of the record of output data.

The abstract machine described is in fact the skeleton for a variety of possible processes on the movement of a three-dimensional distribution of scatterers. The fact that this skeleton has the spatiotemporal form depicted in Fig. 18 that interplays with FSM structures facilitates the user's development of these processes. After the production of the output data indicated in the state diagram of Fig. 55, pages CE contain the original integrated measurements, and the planes of pages TE contain a distribution of apparent movement along each axis.

The interest of carrying out this program lies in the experience we can obtain concerning the use of abstract machines for modeling, and thus programming, the execution of complex data manipulations. In the first place, it has been possible to devise an execution that processes with a minimum of storage the data as soon as they are produced by the radar. It is very apparent from Fig. 55 that the possibility of constructing special processing structures leads to many shortcuts in the execution that are not possible with conventional programming languages. In the second place, it has been found that the development of the program, its readability, and its extension are easier in the language of abstract machines than in a phrase programming language.

The programming language most effective for this type of process is undoubtedly PL-1. A program equivalent to that expressed in Fig. 55 has been developed in PL-1. Starting from the solutions already developed for the abstract machines of Fig. 55, which are applicable only partially to conventional programming, several days were necessary to make the PL-1 program run properly. The source listing of this program contains 115 statements, after exclusion of the input and output parts. The flowchart representing the interconnections among routines includes 31 boxes. The program is given the computer through 159 punch cards, and the optimizing compiler produces an object listing of 859 instructions.

This program supports with clear evidence the belief that there is a practical counterpart to the conjectures of von Neumann (pp 60-61) and Burks (section 2.3.6). In essence, both suggest that for complex activities, the description of a special machine that performs the activity is simpler than a description of the activity itself. The programs shown in this chapter all indicate that a structural description is more effective than a phrase description. For simple processes, however, we can always suspect that this effectiveness is merely caused by the presence or absence of specific features in the language. But as soon as the processes become more complex, the difference in the complexity of the description appears so basic that we have to consider unquestionable that a structural modeling, and thus programing, is more effective, at least for certain classes of problems, than a modeling, and thus programing, by means of a phrase language.

Chapter 7

Concluding Remarks

This research study has had two main objectives: (1) to clarify the reasons why a certain programable processing machine, called the CPL system (section 4.4), has exhibited not only a significant efficiency in the execution but also a conciseness of its machine-language programs that is even greater than that of corresponding high-level language programs, to assess these reasons, to inquire about their possible generality, and to evaluate advantages and disadvantages in respect to other approaches; and (2) to explore the possible applicability of this approach in fields other than the one for which it was originally devised.

The first objective has been carried out to a certain extent in the various chapters of the report, and a summarizing characterization is offered in section 7.1.

In regard to the second objective, this study has broadened the issue, and at the same time has made manifest the need for further study and analysis. On the one hand, the clarification made under task (1) suggests that a structural language is as fully general as a phrase language. Moreover, psychological analyses indicate that a structural domain evokes more easily certain aptitudes of the user, such as familiarity with the spatiotemporal frame, and geometrical intuition. Furthermore, spatiotemporal constructs appear more directly implementable by hardware than are verbal constructs. But on the other hand, a large body of knowledge and experience has been accumulated in the last decades on phrase languages and their implementations, while almost none is available on structural languages. Under the limited means and scope of the present contract, only microscopic bits of

what has been developed in the computer field could be considered. Therefore, in section 7.2 the particular results obtained are reviewed, and their possible extrapolation discussed. And in section 7.3, subjects that deserve further study are suggested.

7.1 THE ESSENCE OF THE APPROACH

The essence of the computer approach analyzed in this report can be presented from various viewpoints. In a computer context, one can say that the structural approach used with analog computers has been brought to a generality and a flexibility similar to those of the programming language approach of digital computers. This result has been achieved by means of a language that describes the structure of the computer.

Analog computers exhibit great efficiency in execution and a reduced need for a memory, and similar characteristics are found in the CPL system. Digital computers free the user from any hardware contact by means of a user language, and a similar facility exists also for the CPL system.

In a programming context, one can say that in this approach the user's images are utilized directly as a form of expression of his intentions. This is achieved by providing formal means (called a symbolic substratum) for describing abstract machines in a spatiotemporal frame. In this substratum there is no gap between the mental images of the user and the actual structures that the computer then will implement.

Often people think of images as vague objects, especially good in an artistic context, while verbal expressions are considered appropriate in a rigorous context. In fact, it has been ascertained (see chapter 2) that images and words are both suitable for both artistic expression and logical structuring. The development of these means of expression and the training of their users affect their applicability.

In the CPL system there is one especially important feature. What is here called a symbolic substratum is in effect a formal language with broad connotations that is capable of implementing phrase structure constructs as well as spatiotemporal constructs. Therefore, when a process, or part of it, is better expressed in a phrase form, such forms also can be interpreted and implemented.

In section 2.1 it was repeatedly observed that an alternation of verbal structures and imagery is a characteristic of mental processes. We can expect that the availability of these two complementary forms in the programming language should facilitate the process of programming and lead in general to more effective programs.

In a formal context, one can say that finite state automata have been given such a flexibility that they constitute here a convenient language for modeling, and thus describing, processes of widely differing natures. This has been achieved by assuming a large alphabet, a variety of formulations for the characterizing functions, and a hierarchical modularity. Because they have been freed from the requirements of being interesting analytical tools, these automata can be given characteristics enabling them to form practical models of processes.

From this approach, one can expect that some sort of systematic guidance can be produced for assisting programming, presently merely empirical. Significant also is the finding that by programming the structures to be implemented, rather than programming the activity of structures already implemented, the complexity of programs appears to be reduced. A relation with the conjecture of von Neumann cited in section 2.5.4, and with the Burks' suggestion (section 2.3.6) is hard to escape.

The reasons that several viewpoints are necessary for better focalizing this approach are undoubtedly the same as those that require several viewpoints for focalizing the notion of automaton. The issue is the communication between man and processing machines; therefore, it should not be surprising that the multiplicity of human mental structures is to be considered.

On page 13 the intent was expressed of exploring with a fresh outlook possible ways of direct communication between man and machine. In the approach taken, the notion of abstract machine appears to be the crucial point. The abstract machines considered here are suitable of rigorous treatment, inasmuch as they can be viewed as automata. This rigorousness is mandatory for dealing with digital machines. Abstract machines evoke the imagery system of the mental processes, which permits a more extended utilization of the user's capabilities. Abstract machines themselves offer the structures to be given the physical computer, making possible a direct implementation of the user's intentions.

When the flexibility and the abstraction of abstract machines are brought to a certain degree, it becomes unclear whether they any longer can be considered machines or whether they should be rather viewed as representation of thought. We are accustomed to consider verbal expressions as "the" representation of thought; but as Arbib (1972) suggests, it seems more appropriate to consider verbal expressions as "samples" of thought. For several classes of activity, it seems that abstract machines are more appropriate samples of the user's thought.

When dealing with a computer, the user's thought must be brought to a rigorous form. Therefore, the user has to use a language, whatever it is, that can finally lead to a precise formulation of the desired activity. The syntax of abstract machines is familiar to all users, regardless of their spoken language or of their professional training. It is the syntax of the spatiotemporal frame that is provided to the user in a natural form by his life experience and sharpened by his geometrical intuition. Moreover, it seems that a structural language is less ambiguous than a verbal language. To avoid ambiguity, a verbal language needs to be so rigid that it becomes a poor language. A structural language seems capable of being at the same time unambiguous and flexible.

7.2 RESULTS OBTAINED AND EXTRAPOLATIONS

In the specific field of real-time data processing in which the CPL 1 machine has been used, and for which the CPL 2 programs have been prepared, the results are clear. With respect to the conventional computer approach, significant advantages are simultaneously obtained in all the aspects involved in the use of a processing machine, as is testified by the examples in chapter 6. It is this multiaspect, always-present facility that has suggested the extent of the study described in this respect.

Ease of prescribing specific implementations of the processes, a paramount requirement in real-time data processing, is very apparent from the flexibility and the isomorphism of the symbolic substratum that constitutes the user's language and the physical substratum that constitutes the computer. In particular, the facility for handling parallelism has been remarkable. The language describes both actual parallelism of different operations on

several variables in a page, and virtual parallelism of many pages performing independent or concurrent tasks.

The speed of execution and the reduced need for memory are easily understandable consequences of the fact that the machine is structured ad hoc for each process. But there is also another consequence from this structuring: economy. When the programability of the hardware reaches the degree implied here, the same hardware can perform the tasks usually accomplished by several different specialized units.

From a practical viewpoint, an unusual characteristic of this approach is the possibility for the user to follow and understand easily all detailed actions of the computer. The benefits are found especially in debugging and modifying programs. Moreover, this characteristic permits a truly man-machine interaction without the need for auxiliary equipment and software systems.

The most striking result is perhaps the fact that the machine programs are typically more concise and more user oriented than the corresponding programs in high-level languages. This outcome is the consequence of structuring and of the isomorphism between the symbolic and the physical substrata.

One objective of this study is to inquire to what extent the above advantages can be applicable to other fields. Here, the extrapolations and considerations that can be made at the present state of the study are presented.

The approach described in this report has, from a theoretical point of view, all the generality required for a general purpose computer. This issue was discussed in section 3.4. From a practical viewpoint, in section 4.3, the favorable characteristics of extended pipelining, flexible parallelism, and minimization of the addressing function were pointed out. In section 5.4 discussions and conjectures were made in regard to the characteristics of the programming language.

Some points can be a priori established. The execution will typically be either more efficient than or equal to that of conventional computers, for the reason that a specialized machine is in general more efficient for a specific task than is a general-purpose machine. For the processes for which the structure of present computers is optimum, this structure can be programmed in the CPL system. Certain inefficiencies encountered in present parallel computers are overcome here because the parallelism can be adjusted to various

degrees. In fact, here, parallelism is a special case of the specialization of the machine.

In regard to cost, no analysis has been carried out. However, the feeling has developed that, in general, a higher utilization of the hardware should occur in comparison to present computers. This should lead to a lower cost, as soon as the new techniques and procedures become "normal".

In section 1.3 several works on computers oriented to programming languages were cited. These are attempts to make a physical structure implement verbal structures. It appears that the level at which this implementation occurs does not notably affect the overall efficiency. In the CPL system, instead, an attempt is made to make a physical structure implement imagery structures. The comparison between CPL 1 and CPL 2 programs suggests that both the ease of programming and the overall efficiency of the execution appear to increase with the level at which a process is modeled in the machine context.

In regard to programming, we can expect general benefits from a structural language, because it is a characteristic of the human mind to have typically greater facility for spatiotemporal frames than for formal systems of verbal structure, even allowing for specific individual characteristics. Any assessment of this question will perhaps be long and controversial, as was the dispute at the beginning of the century over whether thought is imageless or wordless. But such a possible polemic is of no concern to this study, for in psychology it has been clarified that thought uses both verbal and spatiotemporal constructs. And similarly the CPL system is suitable for both verbal and spatiotemporal descriptions.

The present growing interest in structural representations, both in the context of theory of computation and in the context of actual communication with computers, testifies that the present computer approach is too restricted to phrase structures. The crucial point is how to develop a connection between the compiler approach, with all its well-known achievements, and new possible structural approaches. The system described in this report might offer a way for attacking the problem. In the next section, specific topics to be studied are suggested.

7.3 - TOPICS FOR FURTHER STUDY

The introduction of a structural approach in computer programming raises a number of questions that could not be answered within the limited means of this contract - questions that range from the purely theoretical to the specifically pragmatic. A few specific topics that appear most deserving of attention are listed below.

1. Further analysis of programs

The preparation of the programs for real-time processing of radar signals, samples of which are reported in chapter 6, have contributed much to the understanding of a structural approach, and to the development of its programming language. Further benefits both to the understanding of the approach and to the programming language can be expected from analysis of other programs in different fields. Such analyses are fruitful in any of the three ways they can be carried out: (1) by simple preparation of programs in terms of languages of different levels; (2) by computer simulation of different CPL systems; and (3) by actual execution on a CPL machine that can implement a level of sufficient interest.

2. Study of possible connections between the structural approach and the compiler approach

The advantages encountered with the structural approach in the field of applications referred to in chapter 6 are impressive. But no less impressive are the advantages of the compiler approach in many other different fields. Therefore it is natural to inquire whether the two approaches can be used simultaneously. A programmable substratum controlled by finite state automata is well suited also for implementing the activities performed by compilers. Moreover, given the particular flexibility of the FSMs and the self-development of data structures, we can expect that the compiling activity would be particularly efficient.

In this context one can analyze the possibility of embedding the execution with the compilation. The structural approach would be used when appropriate, and the compiler approach when needed. In this case, there would be not two distinct phases, the compilation and the execution, but only the execution, in which, sporadically, different levels of compilation might occur.

Given the time sharing structure of the CPL system, such a fragmentation of compilation and execution activities should not impair the smoothness of the execution. Because of the parallelism that can be implemented in the CPL system, an overall greater efficiency might occur in several cases. Moreover, with real-time compilation, the interaction with the user would be eased.

3. Study of the optimum characteristics for a programable substratum

One of the most intriguing features of the CPL system is the bypassing of the compilation by establishing an isomorphism between the symbolic substratum that constitutes the programming languages and the physical substratum that constitutes the computer. To give the user the impression of not being constrained by the machine, the symbolic substratum should permit the description of any structure that the user is capable of devising. Although such a goal might appear presumptuous, there is good reason for thinking that mental processes are based on a finite set of means, from which structures are continuously being built by grouping and hierarchical concatenation. If a sufficiently close approximation of these means is given to the symbolic substratum, it might be possible to implement a substratum - that is, a language - that appears satisfactorily universal to the user.

One example of this type of study was given in section 5.1.3. However, a much larger scope is necessary. In particular, an analysis of primitives for the data transformation function F would be of great interest. Such a study should concern itself simultaneously with mathematical formulations, psychological aspects of process modeling, and also with conveniences of implementation. Today the situation is different from that of decades ago. When computers developed, technology was posing the most stringent limitations. Today, instead, it is knowledge and imagination that limit the exploitation of technical possibilities.

4. Study on the optimum complexity for a programable network

The larger and the richer in features the programable network, the more likely the user's constructs can be implemented directly; but at the same time the utilization of the investment in hardware decreases. On the side of the programs, in a first approximation, the length of the description of a network

configuration can be related to the logarithm of the number of elements and options in the network. With the increase in number of elements and options, the number of configurations and states necessary to model a process decreases. Therefore we can expect that an increase in the complexity of the programable network will initially decrease the total size of programs. However, with a further increase in network complexity, the increased size of the network descriptions will, statistically, outweigh the reductions in the modeling of the processes.

It appears, therefore, of interest to analyze optimum complexities of programable networks in reference to various criteria such as ease of programming, size of programs, efficiency of execution, and cost-performance ratio.

5. Self-development

As indicated in the Preface, only a preparatory part of the study of self-developing computers could be carried out under this contract. Effort has been concentrated on formulating a programable substratum. In the context of this substratum, self-development can then be studied with a great deal more ease than in the context of conventional computers. Such a study would be of interest not only for application to artificial intelligence and robotics, but also for programming per se. A certain degree of self-development permits an implicit description of processes, with consequently larger flexibility of programs, and reduction of their size.

6. Study of implementation of recursion in the CPL system

Recursion can be implemented without the use of a compiler by giving the processes certain self-developing structures. Certain features of the CPL system appear interesting for such implementations. The flexibility of the transition function, in particular the stopover transitions and the priority, permits the self-construction of recursive paths. The interplay that is possible among FSMs, pages, and data structures in the functional memory give still other possibilities for recursive processes.

7. Study of the effectiveness of the CPL system for list processing

As is well known, certain classes of processing are effectively modeled as a manipulation of lists, a list being a dynamic path in arrays of formal

objects. The CPL system is characterized as a programable substratum; in this substratum, lists of objects can be defined and processed.

There are some features that deserve analysis in the context of list processing. Segments of pages can be created and disposed during processing. Pages can be inserted, deleted, and moved to any point of page segments. The pages are dynamic blocks of data that can vary their size during processing. The page segments are automatically relocated in the memory at each circulation. The merging of pages and FSMs in the programable network allows the data structures to be processing structures also. Specific words in the pages can be used for addressing by means of routing. Data can be transferred from one place to another in the data structures by means of the auxiliary page array Ω'_N . Search can be implemented by one FSM shared by many sequences of pages. Page segments and data structures in the functional memory may work concurrently.

After a certain degree of knowledge has been developed at least in some of the areas indicated above, a more suitable ground will be available for assessing a structural approach and analyzing the possibility for an enlarged general programming language that is capable of both verbal and structural expressions.

R E F E R E N C E S

- Adey, W. R. (1968): "Aspect of cerebral organization information storage and recall" in Corning (1968), p. 69-100.
- Aho, Alfred V. (Ed.) (1973): Currents in the Theory of Computing, Prentice-Hall, Englewood Cliffs, N.J.
- Aiserman, M. A., L. A. Gusev, L. I. Rozonoer, I. M. Smirnova, and A. A. Tal (1971): Logic, Automata, and Algorithms, Academic Press, N.Y. (original in Russian, 1963).
- Akers, Sheldon B. (1972): "A rectangular logic array", IEEE Trans. Comp., CE-1, p. 848-857.
- Alt, Francis L. and Morris Rubinoff (Eds.) (1968): Advances in Computers, Vol. 9, Academic Press, N.Y.
- Amarel, Saul (1970): "On the representation of problems and goal-oriented procedures for computers" in Barnerji and Mesarovic (1970), p. 179-244.
- Anderson, J. P. (1961): "A computer for direct execution of algorithmic languages", Proc. Eastern JCC, Vol. 20, p. 184-193, AFIPS Publ.
- Arbib, M. A. (1968): "Automata theory as an abstract boundary condition for the study of information processing in the nervous system" in Leibovic (1969), p. 3-19.
- Arbib, M. A. (1969): Theories of Abstract Automata, Prentice-Hall, N.J.
- Arbib, M. A. (1972): "Content and consciousness: the secondary role of languages", Report 71 P-1, Computer and Inf. Science, University of Mass., Amherst, Mass.
- Atlas, David (1964): "Advances in radar meteorology", in Advances in Geophysics, Vol. 10, p. 318-478, Academic Press, N.Y.
- Austin, Pauline M. and Spiros G. Geotis (1971): "On the measurement of surface rainfall with radar", Res. Rep. Weather Radar Research, Dept. of Meteor., MIT, Cambridge, Mass.
- Balzer, Robert M. (1973): "Digital computer programming" in Yearbook of Science and Technology, p. 149-151, McGraw-Hill.
- Barnerji, R. and M. D. Mesarovic (Eds.) (1970): "Theoretical approaches to non-numerical problem solving", Proc. IV Systems Symp. (1968), Springer-Verlag, Berlin.
- Barrow, D. W. and C. Strachey (1966): "Programming" in Advances in Programming and Non-Numerical Computation (Ed. L. Fox), p. 49-82, Pergamon Press, N.Y.
- Bartee, T. C., I. L. Lebow, and I. S. Reed (1962): Theory and Design of Digital Machines, McGraw-Hill, N.Y.
- Bashkow, T. R., A. Sasson, and A. Kronfeld (1967): "System design of Fortran machine", IEEE Trans., EC-16, p. 485-499.
- Battan, Louis J. (1959): Radar Meteorology, Univ. of Chicago Press.

- Bauer, W. F. (1960): "Horizons in computer system design", Proc. Western JCC, p. 41-52.
- Berge, Claude (1968): The Theory of Graphs, Wiley, N.Y.
- Bell, C. Gordon and Allen Newell (1971): Computer Structures: Readings and Examples, McGraw-Hill, N.Y.
- Beth, E. W. and J. Piaget (1966): Mathematical Epistemology and Psychology, Reidel Publishing Co., Dordrecht, Holland.
- Bjorner, D. (1971): "On the definition of higher-level language machines", Proc. Symp. Computer and Automata, Polytechnic Press, N.Y., p. 105-135.
- Blinkov, Samuil M. and Il'ya I. Glezer (1968): The Human Brain in Figures and Tables, Basic Books, Inc., Plenum Press.
- Blum, Manuel (1967): "A machine-independent theory of the complexity of recursive functions", Journal ACM, Vol. 14, p. 322-336.
- Blum, Manuel (1967): "On the size of machines", Information and Control, Vol. 11, p. 257-265.
- Blumenthal, A. L. (1970): Language and Psychology, Historical Aspects of Psycholinguistics, John Wiley & Sons, Inc.
- Book, Ronald V. (1973): "Topics in formal language theory" in Aho (1973).
- Booth, T. L. (1967): Sequential Machines and Automata Theory, John Wiley & Sons, N.Y.
- Bruner, J. S., R. R. Oliver, P. M. Greenfield, et al (1966): Studies in Cognitive Growth, John Wiley & Sons, N.Y.
- Bucholz, W. (Ed.) (1962): Planning a Computer System, McGraw-Hill, N.Y.
- Burkhardt, W. H. (1965): "Universal Programming Languages and Processors, a brief survey and new concepts", AFIPS, Fall JCC, Vol. 27, part 1, p. 1-21.
- Burks, A. W. and Jesse B. Wright (1953): "Theory of logical nets", Proc. IRE, Vol. 41, p. 1357-1365.
- Burks, A. W. (1963): "Programming and the theory of automata" in Braffort and Hirschberg (Eds.), Computer Programming and Formal Systems, p. 100-117, North-Holland Publ. Co., reprinted in Burks (1970).
- Burks, A. W. (1970): Essays on Cellular Automata, Univ. of Illinois Press, Urbana, Ill.
- Caianello, E. R. (1961): "Outline of a theory of thought processes and thinking machines", J. Theor. Biol., 1, p. 204-235.
- Cantarella, A., F. Maffioli, and A. Pawlina (1971): "On the sirio SHF experiment and weather radar data handling", Symp. Satel. Com., Geneva, Italy, June 1971.
- Cappetti, Ilio and Mario Schaffner (1972): "Structure of a communication network and its control computers", Proc. Symp. Computer-Communications Networks and Teletraffic, p. 587-598, Polytechnic Press, Brooklyn, N.Y.
- Chaitin, Gregory J. (1966): "On the length of programs for computing finite binary sequences", Journal ACM, Vol 13, N4, p. 547-569.

- Chen, T. C. (1971): "Parallelism, pipelining and computer efficiency", Comp. Res., Vol. 10, p. 69-74.
- Chesley, G. D. and W. A. Smith (1971): "The hardware-implemented high-level machine language for SYMBOL", Proc. Spring JCC, Vol. 38, p. 563-573, AFIPS Press, N.J.
- Chomsky, N. (1957): Syntactic Structures, The Hague: Mouton & Co.
- Codasyl (1962): "An information algebra", Com. ACM, Vol. 5, p. 190-
- Codd, E. F. (1968): Cellular Automata, Academic Press.
- Comfort, W. T. (1962): "Highly parallel machines", Proc. Workshop on Computer Organization, p. 126-155 (1963).
- Comfort, W. T. (1963): "A modified Holland machine", Proc. Fall JCC, p. 481-488.
- Cooke, I. and M. Lipkin, Jr., (Eds.) (1972): Cellular Neurophysiology - a source book, Holt, Rinehard and Winston, Inc., NY.
- Corning, W. and M. Balabam (Eds.) (1968): The Mind: Biological Approaches to Its Functions, Interscience Publisher (John Wiley), N.Y.
- Cotten, L. W. (1969): "Maximum-rate pipeline systems", Proc. Spring JCC, p. 581-586.
- Dennis, J. B. (1971): "On the design and specification of a common base language", Proc. Symp. Computers and Automata, p. 47-74, Polytechnic Press, N.Y.
- Dertouzos, Michael L., Michael Athams, Richard N. Spann, and Samuel J. Mason (1972): Systems, Networks and Computation: Basic Concepts, McGraw-Hill, N.Y.
- Dijkstra, Edsger W. (1968): "Co-operating sequential process" in Programming Languages (Genus editor), p. 43-109, NATO Symposium.
- Eccles, J. C. (1964): "Conscious experience and memory", in Eccles (1965), p. 314-344.
- Eccles, J. C. (Ed.) (1965): Brain and Conscious Experience, Springer-Verlag, N.Y.
- Estrin, G. (1960): "Organization of a computer system - the fixed plus variable structure computer", Proc. Western JCC, p. 33-40.
- Estrin, G., B. Russell, R. Turn, and I. Bibb (1963): "Parallel processing in a restructurable computer system", IEEE Trans. Comp., Vol EC-12, N5, p. 747-755.
- Falkoff, A. D. and K. E. Iverson (1967): "APL 360 terminal system", Proc. Symp. Interactive System, Academic Press.
- Fields, W. S. and W. Abbott (Eds.) (1963): "Information storage and neural control", Proc. Houston, Texas, X Annual Scientific Meeting, Neurological Society.
- Flavell, J. H. (1963): The Developmental Psychology of Joan Piaget, Van Nostrand Co., Inc., Princeton, N.J., 422 pp.
- Fleisher, Aaron (1953): "The information contained in weather noise", Res. Rep. N22, Dept. of Meteor., MIT, Cambridge, Mass.

- Flynn, M. J., (1972): "Some Computer Organizations and their Effectiveness", IEEE Trans. C-21, pp948-960.
- Furth, H. G. (1969): Piaget and Knowledge, Prentice-Hall, New Jersey.
- Galler, B. A. and A. J. Perlis (1970): A View of Programming Languages, Addison-Wesley, Reading, Mass.
- Gardner, M. (1958): Logic Machines and Diagrams, McGraw-Hill, New York
- Gill, A. (1962): Introduction to the Theory of Finite-State Machines, McGraw-Hill, New York.
- Ginsburg, S. (1962): An Introduction To Mathematical Machine Theory, Addison-Wesley, Reading, Mass.
- Ginsburg, S. (1966): The Mathematical Theory of Context Free Languages, McGraw-Hill, New York.
- Gold, Oppenheim, Rader, and Stockholm (1969): Digital Processing of Signals, McGraw-Hill, New York.
- Goldstine, H. H. and J. von Neumann (1947): "Planning and Coding Problems for an Electronic Computing Instrument", reprinted in A. H. Taub, John von Neumann Collected Works", Vol. V, Pergamon Press, 1961, pp. 80-235.
- Graham, W. R., (1970): "The Parallel and the Pipeline Computers", Datamation, Vol. 16, April, 1970.
- Grosky, W. I. and F. Tsui (1973): "Pattern Generation in Non-Standard Tessellation Automata", Proc. ACM Annual Conf. pp. 345-348.
- Harrison, M.A. (1965): Introduction to Switching and Automata Theory, McGraw-Hill, New York.
- Hartmanis, J. and R. E. Stearns (1966): Algebraic Structure Theory of Sequential Machines, Prentice-Hall, New Jersey.
- Hassitt, A., J. W. Lageschulte and L. E. Lyon (1973): "Implementation of a High Level Language Machine", Com. ACM Vol. 16, pp. 199-212.
- Hellerman, Leo, (1972): "A Measure of Computational Work", IEEE Trans. Comp. Vol. C-21, N5 pp. 439-446. Follow-up in IEEE Trans Comp. Vol.
- Hennie, F. C. (1968): Finite-State Models for Logical Machines, John Wiley and Sons, Inc., New York.
- Hobbs, L. C., and al. (Ed.), (1970): Parallel Processor Systems, Technologies, and Applications, Spartan Books, New York.
- Holland, J. H. (1959): "A Universal Computer Capable of Executing an Arbitrary Number of Subprograms Simultaneously", Proc. Eastern J.C.C., pp. 108-113; reprinted also in Burks (1970).
- Holland, J. H. (1960): "Iterative Circuit Computers", Proc. Western J.C.C. pp. 259-265; reprinted in Burks (1970).
- Holland, J. H. (1965): "Iterative Circuit Computer Characterisation and Resume of Advantages and Disadvantages", in Mathis et. al. (Ed.) Microelectronics and Large Systems, pp. 171-178, Spartan Books, New York; reprinted in Burks (1970).

- Holland, J. H. (1970): "Hierarchical Descriptions, Universal Spaces, and Adaptive Systems", in Burks (1970), pp. 320-353.
- Holt, Anatol W. (1971): "Introduction to Occurrence Systems", in Associative Information Techniques, edited by Jacks (1971), pp. 175-203.
- Hopcroft, J. E. and J. D. Ullman (1967): "An Approach to a Unified Theory of Automata", Bell System Tech Journal, V. 46, pp. 1793-1829.
- Huffman, D. A. (1954): "The Synthesis of Sequential Switching Circuits", Journal of the Franklin Institute, 257, pp. 161-190-275-303.
- Iliffe, J. K. (1968): Basic Machine Principles, American Elsevier Publ.
- Iliffe, J. K. (1969): "Elements of BLM", The Computer Journal, Vol. 12, pp. 251-2581.
- Inhelder, B. and J. Piaget (1958): The Growth of Logical Thinking/Construction of Formal Operational Structures, Basic Books.
- Iverson, K. E. (1962a): A Programming Language, John Wiley and Sons, New York.
- Iverson, K. E. (1962b): "A Common Language for Hardware, Software, and Applications", Proc. Fall JCC, Vol. 22, p. 121.
- Iverson, K. E. (1964): "Formalism in Programming Languages", Com. ACM, Vol. 7, n. 2, pp. 80-88.
- Jacks, Edwin L. (ed.) (1971): "Associative Information Techniques", Proc. Symp at General Motors Research Laboratory, American Elsevier, New York.
- Jump, J. Robert and Fritzsche, Dennis R. (1972): "Microprogrammed Arrays", IEEE Trans. Comp. Vol. C-21, pp. 974-984.
- Karp, R. M. and Miller, R.E. (1967): "Parallel Program Schemata: A Mathematical Model for Parallel Computation", IEEE Conf. Rec. 8th Symp on Switching and Automata Theory, pp. 55-61.
- Kavanagh, T. F. (1960) "Tabsof, A Fundamental Concept for System-Oriented Languages", Proc. 1960 Eastern JCC, pp. 117-136.
- King, C. A. (1972): "A Graph-Theoretic Programming Language" in Read (1972) pp. 63-75.
- Kleinmuntz, B. (ed) (1966): Problem Solving: Research, Method and Theory, John Wiley and Sons, New York.
- Kobrinskii, N.E. and B. A. Trakhtenbrot (1965): Introduction to the Theory of Finite Automata, North-Holland Publ., Amsterdam, Holland.
- Koczela, L. J. (1968): The Distributed Processor Organization, in Alt and Rubinoff (1968).
- Kutti, A. K. (1928): "On a Graphical Representation of the Operating Regime of Circuits", in E. F. Moore (Editor), Sequential Machines, Selected Papers, Addison-Wesley Publishing Co., Inc., 1964, translated from "Trudy Leningradskoi Eksperimental noi Elektrotekhnicheskoi Laboratorii," Vol. 8 (1928).
- Leibovic, K. N. (1969): Information Processing in the Nervous System, Springer-Verlag, New York.

- Lettvin, J. Y., H. R. Marturana, W. S. McCulloch, and W. H. Pitts (1959): "What the Frog's Eye Tells the Frog's Brain", Proc. IRE, Vol. 47, pp. 1940-1951; reprinted in Corning (1968), pp. 233-258.
- Longuet-Higgins, H. C. (1969): "The Non-local Storage and Associative Retrieval of Spatio-Temporal Patterns" in Leibovic (1969), pp. 37-47.
- Low, D. W. (1973): "Programming by Questionnaire: An Effective Way to Use Decision Tables" Com. ACM, Vol. 16, pp. 282-286.
- Luce, R. Duncan, Robert R. Bush and Eugene Galanter (Editors) (1963): Handbook of Mathematical Psychology, John Wiley and Sons.
- Luria, A. R. (1966): Higher Cortical Functions in Man, Basic Book, New York (original in Russian, 1962).
- MAC (1970): Record of the Project MAC Conf. on Concurrent Systems and Parallel Computation, Published by ACM.
- Mandler, J. M., Mandler, G. (1964): Thinking, from Association to Gestalt, John Wiley & Sons, Inc.
- Manna, Zohar (1973): "Program Schemas" in Aho (1973), pp. 90-142.
- Maruoka, Akira and Honda, Namio (1973): "Logical Networks of Flexible Cells", IEEE Trans Comp. Vol C-22, N. 4, pp. 347-358.
- McConnell (1968): The Modern Search for the Engram, in Corning (1968), p. 49-68.
- McCulloch, W. S., Pitts, W. (1943): "A Logical Calculus of the Ideas Immanent in Nervous Activity", Bull. Math. Biophys., 5, pp. 115-133, also reprinted in Fields-Abbott (1963), and Moore (1966).
- McCulloch, W. S. (1965): Embodiments of Mind, M.I.T. Press, Cambridge, Mass.
- McDaniel, H. (1970): Decision Table Software, Brandon Systems, Inc. New York.
- McKeeman, W. M. (1967): "Language Directed Computer Design", Proc. Fall JCC, Vol. 31, pp. 413-417.
- Mealy, G. H. (1955): "A Method for Synthesizing sequential Circuits", Bell Systems Technical Journal, Vol. 5, pp. 1045-79.
- Melbourne, A. J. and J. M. Pugmire (1966): "A Small Computer for the Direct Processing of FORTRAN Statements", The Computer Journal, Vol. 8, pp. 24-27.
- Meo, Angelo Raffaele (1968): "Modular Tree Structures", IEEE Trans Comp., Vol. C-17, N. 5, pp. 432-442.
- Meyer, A. R. and M. J. Fischer (1971): "Economy of Description by Automata, Grammars, and Formal Systems", 12th Symp. Switching and Automata Theory, pp. 188-191.
- Miller, G. A. (1964): Mathematics and Psychology, John Wiley & Sons,
- Minnick, Robert C. (1967): "A Study of Micro-cellular Research", Jour. Ass. Comp. Mach. Vol. 14, April 1967, pp. 203-241.

- Minsky, M. (1962): "Problem of Formulation for Artificial Intelligence", Proc. XIV Symp. Applied Math. Amer. Math. Soc., Providence, R. I., pp. 35-46.
- Minsky, M. (1965): "Matter, Mind and Models" Proc. IFIP Congress 65, pp. 45-49, Spartan Books, Washington, D.C.
- Minsky, M. L. (1967): Computation: Finite and Infinite Machines, Prentice-Hall, Inc., Englewood Cliffs, New Jersey.
- Minsky, M. and S. Papert (1969): Perceptrons, the M.I.T. Press, Cambridge, Mass.
- MIT (1968): Research Laboratory of Electronics, Quarterly Progress Report No. 88 (January 15, 1960) pp. 337-339, by R. Moreno-Díaz and W.S. McCulloch.
- Moore, E. F. (1956): "Gedanken-Experiments on Sequential Machines", in Automata Theory (Annals of Mathematics Studies No. 34), Princeton, New Jersey, pp. 129-153.
- Moore, E. F. (ed.) (1964): Sequential Machines-Selected Papers, Adison-Wesley, Reading, Mass.
- Mullery, A. P., R.F. Schauer, and R. Rice (1963): "ADAM-A Problem-Oriented Symbol Processor", Proc. Spring JCC., Vol. 23, pp. 367-380.
- Mullery, A.P. (1964): "A Procedure-Oriented Machine Language", IEEE Trans. EC-13, pp. 449-455.
- Naur, Peter (ed.) (1960): "Report on the Algorithmic Language ALGOL 60", Com. ACM, 1960, pp. 299-314.
- Naur, P. (1965): "The Place of Programming in a World of Problems, Tools and People", Proc. IFIP Congress 65, Spartan Books, Washington, D.C., pp. 195-199.
- Nelson, R.J. (1968): Introduction to Automata, John Wiley and Sons, New York.
- Newell A. (1970): "Remarks on the Relationship between Artificial Intelligence and Cognitive Psychology", in Banerji and Mesarovic (Eds.) Theoretical Approaches to Non-Numerical Problem Solving, Springer-Verlag, Berlin, pp. 363-400.
- Pager, David (1973): "On the Problem of Communicating Complex Information", Com. ACM, Vol. 16, N. 5, pp. 275-281.
- Paivio, A. (1971): Imagery and Verbal Processes, Holt, Rinehart and Winston, Inc. New York.
- Penfield, W. (1965): Speech, Perception and the Uncommitted Cortex, in Eccles (ed.) 1965, pp. 217-237.
- Piaget, J. (1950): The Psychology of Intelligence, Routled & Kegan Paul, Ltd., London.
- Piaget, J. (1971): Biology and Knowledge, University of Chicago Press, Chicago (original in French, 1967).
- Piaget, J. and B. Inhelder (1971): Mental Imagery in the Child-a study of the development of imaginal representation, Basic Books, New York, (original in French, 1966).

- Porter, R. E. (1960): "The RW-400 - a New Polymorphic Data System", Datamation, Vol. 6, N. 1, pp. 8-16.
- Ramarmoorthy, C. V., and S. S. Reddi, "Towards a Theory of Pipeline Computing Systems", Proc. 10th Allerton Conf. on Circuit and System Theory, University of Illinois, Urbana, Ill., pp. 759-568.
- Read, R.C. (ed.) (1972): Graph Theory and Computing, Academic Press, New York.
- Reiss, R.F. (1962): "An Abstract Machine Based on Classical Association Psychology", Proc. Spring JCC, pp. 53-70.
- Reiss, R. F. (1964): "A Theory of Resonant Networks", in Reiss (ed.) Neural Theory and Modelling, Stanford University Press, Stanford, Cal., pp. 427.
- Rosen, R. (1969) "Hierarchical Organisation in Automata Theoretic Models of the Central Nervous System", in Leibovic (1969), pp. 21-35.
- Rosenblatt, F. (1962): Principles of Neurodynamics, Spartan Books.
- Rosenstiehl, P., J.R. Fiksel, and A. Holliger (1972): "Intelligent Graphs: Networks of Finite Automata Capable of Solving Graph Problems", in Read (ed.) (1972), pp. 2.9-265.
- Sammet, J. E. (1969): Programming Languages: History and Fundamentals, Prentice-Hall, Inc. New Jersey.
- SAO (1966) "Measurement of High Altitude Atmospheric Parameters by Radar Meteor Techniques", Final Report, AF 19(628)-3248, Smithsonian Astrophysical Observatory, Cambridge, Mass., Vol. III, Appendix, Chap. 8 (Schaffner).
- Schaffner, M. R. (1964): "The Ensemble Digital Processor", NASA-158 Tech. Rep. No. 1, Harvard College Observatory, Cambridge, Mass.
- ____ (1966): "The Circulating Page Loose System. A New Solution for Data Processing." Harvard-Smithsonian Radio Meteor Project Res. Rep. No. 15.
- ____ (1971): "A Computer Modeled after an Automaton", Proc. Symp. Computers and Automata, MRI Symp. Vol. XXI, p. 635-650, Polytechnic Press, Brooklyn, N. Y.
- ____ (1972a): "Computers Formed by the Problems rather than Problems Deformed by the Computers", Digest 6th IEEE Int. Computer Conf., p. 259-264, San Francisco, Cal.
- ____ (1972b): "On the Data Processing for Weather Radar", Prepr. 15th Radar Meteorology Conf., Amer. Meteor. Soc., Boston, Mass. pp. 368-373.
- ____ (1972c): "Echo Movement and Evolution from Real-Time Processing", Prepr. 15th Radar Meteorology Conf., Amer. Meteor. Soc., Boston, Mass. pp. 374-378.
- ____ (1972d): "A Procedure for Describing Discrete Processes", Proc. 10th Allerton Conf. on System and Circuits Theory, Univ. of Ill. Urbana, Illinois, pp. 256-282.

- (1973): "A Computer Architecture and its Programing Language", Proc. 1st Symp. Comp. Arch. Univ. of Florida, Gainesville, Florida, pp. 271-277.
- Schlesinger, S. and L. Saskin (1967): "POSE, A Language for Posing Problems to a Computer", Com. ACM, Vol. 10, n. 5., pp. 279-285.
- Schmitt, F. O. (1968): "Molecular Correlation of Brain Functions", in Gruing (1968), pp. 23-47.
- Schwartz, Jules I. (1965): "Comparing Programming Languages", Computers and Automation, Feb. 1965, p. 15.
- Shannon, C.E., and J. McCarthy (ed.) (1956): Automata Study, Princeton University Press, Princeton, New Jersey.
- Shaw, Christofer J. (1966): "Assemble or Compile?" Datamation, Vol. 12, No. 9, pp. 59-62.
- Sheldon, B. Akers (1972): "A Rectangular Logic Array", IEEE Trans, Comp. Vol. C-21, pp. 848-857.
- Slotnick, D.L. (1962): "The Solomon Computer", Proc. Fall JCC, pp. 97-107.
- Slutz, D.R. (1968): "The Flow Graph Schemata Model of Parallel Computation", M.I.T. Project MAC, MAC-TR-53, Cambridge, Mass.
- Smith, W. R. R. Rice, Chesley GD, et al. (1971): "SYMBOL - a large experimental system exploring major hardware replacement of software", Proc. Spring JCC, AFIPS Vol. 39., p. 601-616.
- Steel, T.B. (1961a): "A First Version of UNCOL", Proc. Western JCC, pp. 371-377.
- Steel, T. B. (1971b): "UNCOL the Myth and the Fact", Annual Review Automatic Programming, Vol. 2, Pergamon Press, pp. 325-344.
- Strong, J., Wegstein, J., Tritter, A., Olsstin, J., Mock, O., and Steel, T. (1958): "The Problem of Programming Communication with Changing Machines, A Proposed Solution", Com. ACM, Vol. 1, n. 8 and 9.
- Thomas, Robert H., (1971): "A Model for Process Representation and Synthesis", TR-87, Project MAC, M.I.T., Cambridge, Mass.
- Thurber, K.J. and J.W. Myrna (1970): "System Design of a Cellular APL Computer", IEEE Trans. Comp. Vol. C-19, p. 291-303.
- Turing, A.M. (1936): "On Computable Numbers with an Application to the Entscheidungsproblem", Proc. London Math. Soc., Ser. 2-42, pp. 230-265.
- Turing, A.M. (1950): "Computing Machinery and Intelligence", Mind, Vol. 59, pp. 433-460, reprinted in Feigenbaum and Feldman (ed.) Computers and Thought, McGraw-Hill, New York, pp. 11-35.
- Unger, S.H. (1958): "A Computer Oriented Toward Spatial Problems", Proc. IRE, Vol. 46, pp. 1744-1750.
- Vineberg, M.B. and A. Avizienis (1972): "Implementation of a High-Level Language on an Array Machine", Digest 6th IEEE Comp. Conf. pp. 37-39.
- von Neumann, J. (1948): "The General and Logical Theory of Automata", Hixon Symposium, Pasadena, Calif., reprinted in A. H. Taub, John von Neumann Collected Works, Vol. V, Pergamon Press, pp. 288-328.

- von Neumann, J. (1958): The Computer and the Brain, Yale Univ. Press, New Haven, Conn.
- von Neumann, J. (1966): Theory of Self-Reproducing Automata, Univ. of Ill. Press, Urbana, Ill.
- Wang, H. (1957): "A Variant to Turing's Theory of Computing Machines", Journal ACM, Vol. 4, pp. 63-92.
- Weber, H. (1967): "A Microprogrammed Implementation of EULER on IBM System/360 Model 30", Com. ACM, Vol. 10, pp. 549-558.
- Whorf, B.L. (1956): Language, Thought and Reality, The M.I.T. Press, Cambridge, Mass.
- Winograd, Terry (1972): Understanding Natural Language, Accademic Press, N. Y.
- Yamada, H. and S. Amororo (1969): "Tessellation Automata", Information and Control, Vol. 14, pp. 299-317.
- Yamada, H. and S. Amororo (1971): "Structure and Behavioral Equivalence of Tessellation Automata" Information and Control, Vol. 18, pp. 1-31.